CHAPTER

PERFORMANCE ISOLATION IN CLOUD-BASED BIG DATA ARCHITECTURES

8

Bedir Tekinerdogan[*], **Alp Oral**[*,†]

[*] *Wageningen University, Information Technology, Wageningen, The Netherlands* [†] *Microsoft, Vancouver, Canada*

8.1 INTRODUCTION

In the last decade, the ability to capture and store vast amounts of structured and unstructured data has grown at an unprecedented rate. Traditional data management techniques and tools did not scale with the generated mass scale of data and the need to capture, store, analyze, and process this data within acceptable time [10]. To address these problems, the term big data has been introduced which can be explained according to three V's: Volume (amount of data), Velocity (speed of data), and Variety (range of data types and sources). Big data has now become a very important driver for innovation and growth for various industries such as health, administration, agriculture, defense, and education. To cope with the problems of rapidly increasing volume, variety and velocity of the generated data novel technical capacity and the infrastructure has been developed to aggregate and analyze big data. One of the important approaches is the integration of cloud computing with big data. Big data is now often stored on a distributed storage based on cloud computing rather than local storage. Cloud computing is based on services that are hosted on providers over the Internet. Hereby, services are fully managed by the provider, whereas consumers can acquire the required amount of services on demand, use applications without installation and access their personal files through any computer with Internet access. Cloud computing provides a powerful technology for data storage and data analytics to perform massive-scale and complex computing. As such, cloud computing eliminates the need to maintain expensive computing hardware, dedicated storage, and software applications.

A typical cloud-based big data system has many different tenants [7], which require access to the server's functionality. In a *nonisolated* cloud system, the different clients can freely use the resources of the BCS. Hereby, disruptive tenants who exceed their limits can easily cause degradation of performance of the provided services for other tenants. To meet performance demands of the multiple tenants and meet fairness criteria, various performance isolation approaches have been introduced including artificial delay, round robin, blacklist, and thread pool. Each of these performance isolation approaches adopts different strategies to avoid the performance interference in case of multiple concurrent tenant needs.

In this paper, we propose a framework and a systematic approach for performance isolation in cloud-based big data systems. To this end, we present an architecture design [18] of cloud-based big

Software Architecture for Big Data and the Cloud. DOI: 10.1016/B978-0-12-805467-3.00008-9

data systems and discuss the integration of feasible performance isolation approaches. We evaluate our approach using *PublicFeed*, a social media application that is based on a cloud-based big data platform.

The remainder of the paper is organized as follows. In Section 8.2, we provide the background on cloud computing, and big data architecture. Section 8.3 presents the case study with the problem statement. Section 8.4 presents the state-of-the-art on performance monitoring in cloud-based big data systems. Section 8.5 describes the application framework that integrates performance monitoring with big data systems. In Section 8.6, provides the evaluation of the proposed framework and the approach. Section 8.7 presents the discussion. Section 8.8 presents the related work and finally Section 8.9 concludes the paper.

8.2 BACKGROUND

8.2.1 CLOUD COMPUTING

In general, three types of cloud computing models are defined [16,17,15]. These are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The IaaS model shares hardware resources among the users. Cloud providers typically bill IaaS services according to the utilization of hardware resources by the users. The PaaS model is the basis for the computing platform based upon hardware resources. It is typically an application engine similar to an operating system or a database engine, which binds the hardware resources (IaaS layer) to the software (SaaS layer). The SaaS model is the software layer, which contains the business model. In the SaaS layer, clients are not allowed to modify the lower levels such as hardware resources and application platform. Clients of SaaS systems are typically the end-users that use the SaaS services on-demand basis. We can distinguish between thin clients and rich clients (or thick/fat clients). A thin client is heavily dependent on the computation power and functionality of the server. A rich client is a computer that provides itself rich functionality independent of the central server.

In principle, SaaS has a multitier architecture with multiple thin clients. In Fig. 8.1 the multiplicity of the client nodes is shown through the asterisk symbol (∗). In SaaS systems the thin clients rent and access the software functionality from providers on the Internet. As such the cloud client includes only one-layer User Layer which usually includes a web browser and/or the functionality to access the web services of the providers. This layer includes, for example, data integration and presentation. The SaaS providers usually include the layers of Distribution Layer, Presentation Layer, Business Service Layer, Application Service Layer, Data Access Layer, Data Storage Layer, and Supporting Service Layer.

Distribution Layer defines the functionality for load balancing and routing. *Presentation Layer* represents the formatted data to the users and adapts the user interactions. The *Application and Business Service Layer* represents services such as identity management, application integration services, and communication services. *Data Access Layer* represents the functionality for accessing the database through a database management system. *Data Storage Layer* includes the databases. Finally, the *Supporting Service Layer* includes functionality that supports the horizontal layers and may include functionality such as monitoring, billing, additional security services, and fault management. Each of these layers can be further decomposed into sub-layers.

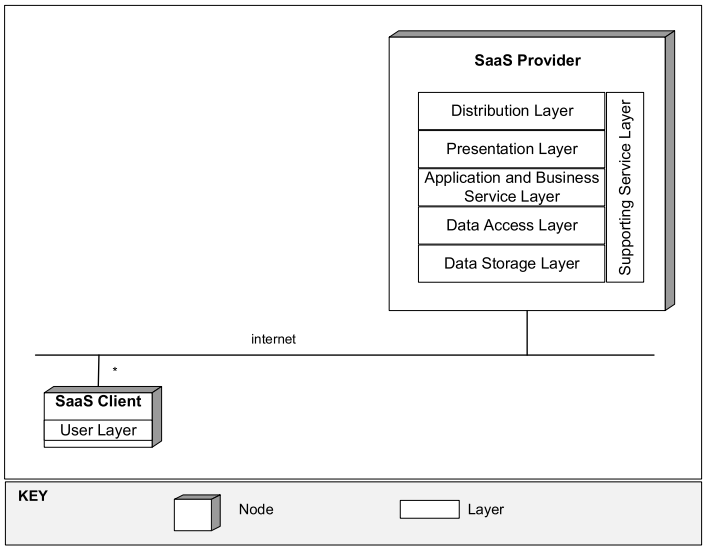


FIGURE 8.1

SaaS reference architecture

8.2.2 BIG DATA ARCHITECTURE

Obviously, an appropriate big data architecture design will play a fundamental role to meet the big data processing needs. Several reference architectures are now being proposed to support the design of big data systems. In this paper, we will adopt the Lambda architecture as defined by Marz [10]. The Lambda architecture is a big data architecture that is designed to satisfy the needs for a robust system that is fault-tolerant, both against hardware failures and human mistakes. Hereby it takes advantage of both batch- and stream-processing methods. In essence, the architecture consists of three layers including batch processing layer, speed (or real-time) processing layer, and serving layer. (See Fig. 8.2.)

The batch processing layer has two functions: (1) managing the master dataset (an immutable, append-only set of raw data), and (2) to precompute the batch views. The master data set is stored using a distributed processing system that can handle very large quantities of data. The batch views are generated by processing all available data. As such, any errors can be fixed by recomputing based on the complete data set, and subsequently updating existing views.

The speed layer processes data streams in real time and deals with recent data only. In essence, there are two basic functions of the speed layer: (1) storing the real time views and (2) processing the incoming data stream so as to update those views. It compensates for the high latency of the batch layer to enable up-to-date results for queries. The speed layer's view is not as accurate and complete as the ones eventually produced by the batch layer, but they are available almost immediately after data is received.

The serving layer indexes the batch views so that they can be queried in low-latency, ad-hoc way. The query merges result from the batch and speed layers to respond to ad-hoc queries by returning precomputed views or building views from the processed data.

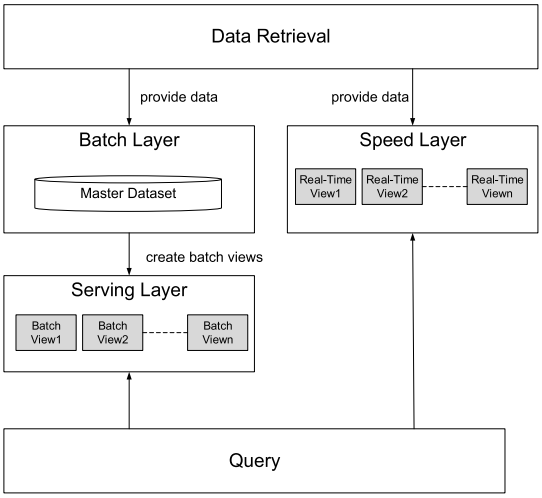


FIGURE 8.2

Lambda architecture for a big data system (adapted from [10])

8.3 CASE STUDY AND PROBLEM STATEMENT

In this section, we describe the case study that is used to illustrate the problem statement.

8.3.1 CASE STUDY

PublicFeed (https://www.publicfeed.com/) is a location and interest based real time social collaborative newsfeed application which has to deal with a variety types of data. The conceptual architecture for PublicFeed is shown in Fig. 8.3.

The architecture represents a cloud-based big data system in which multiple tenants access the big data and the related services via the cloud server. Tenants make requests to fetch results as a data response. In the system, users can register to the system, follow other members in the system, publish feeds, and interact with each other. Registered users are able to publish text feeds, videos, photos and audio files. Feeds have different type of meta-data including uniquely addressing global coordinate, a category, tags, and one of the five different levels of sharing audience selected by Author. Sharing audiences are distinguished based on Country, City, District, Neighbor, and Street levels. System users can real-time follow the feeds for the selected audience level. The selected audience level also covers the subaudience levels such as cities covers districts, neighbors, and street levels. Therefore, the overall data flow in the system is changing for each incoming feed. Given this context, the system can be largely characterized as a big data system from the earlier defined three perspectives of volume, variety, and velocity of the data. The following metric values provide insight in the complexity and size of the current system:

- Total number of users: 212,470

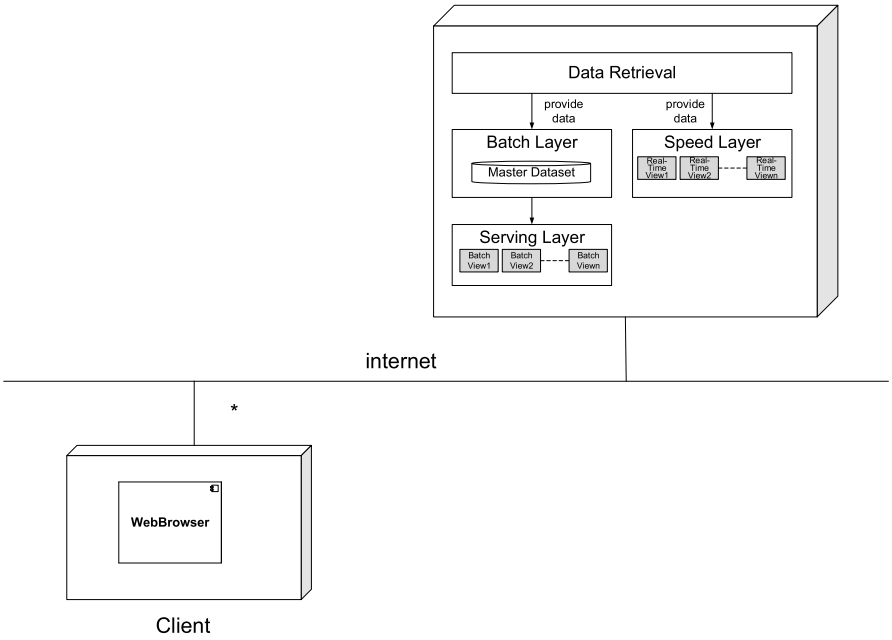


FIGURE 8.3

Conceptual architecture of PublicFeed, a cloud-based big data system

- Total number of registered mobile devices: 61,431
- Total number of different feeds: 147,762
- Total number of different location coordinates assigned to feed: 21,228
- Daily incoming news feeds: 400–1500
- Daily private messages: 5000–20,000
- Daily like/dislike: 3000–9000

The data is generated from end users who wish to share their experiences. The feeds are considered as raw data and further processed. An important information that is extracted is, for example, the *best feeds* that is calculated based on the number of likes. Feed specific interactions are the basic measures for determining the quality of an individual feed. In addition to these interaction data, time of the taken actions also play a key role for this measurement. Individual feed quality affects the overall application, whereas other interactions only affect individual users. An important aspect of the system is the *feed timeline*, which lists 150 feeds, including the user interest, time and sharing audience metrics. Feed timeline is specifically defined for each registered user who has different interests. Besides of the interaction among the users, the system can use a push based notification system that sends information to the clients at specific times and time intervals. Most of the time, this notification is based on other users' actions and is sent to affected users only, but sometimes a broadcast of a specific feed with editorial content might be send to all users. This editorial content is often available due to an unexpected

event – for example, in case of an earthquake or any other important event. These notifications need to be instantly sent and cannot be scheduled.

8.3.2 PROBLEM STATEMENT

In the previous case study, the massive number of users together with their interactions can easily result in performance problems. If the actions of users are not perfectly isolated, the system might soon have to cope with disruptive users, who exceed their limits and thereby cause suffering to the regular users who demand the service within their limits. This situation can lead to a loss of trust in the system, which can eventually lead to a business critical situation. Thus, handling the performance problem in a fair way is as important as providing the functional services of the application.

Sometimes it is possible to increase or decrease the performance for some users, as long as this will not violate other customers' minimum service level agreement (SLA) requirements. An extreme solution is the isolation of computational resources for different users. Hereby, the interference of different users is avoided using isolated resource sharing. In practice, this is not a cost-effective solution since several resources can become unnecessary idle. To cope with performance problems two seemingly feasible solutions could be identified caching, and horizontal scalability.

In the case of caching, frequently accessed data is often cached to shorten data access times, reduce latency, and improve input/output. Since the system workload is dependent upon I/O operations, caching as such can be indeed used to improve application performance. Unfortunately, caching is not always feasible and cannot completely solve the problem of fairness. This is in particular the case if the need for cache updates exceeds the required updates of the actual data. For example, in the case study the feed timeline needed to be updated at real time and was based on both individual user interaction and other incoming related feeds. In this situation, the frequency of cache update can occur more than the timeline requests. Hence, caching does not solve the performance and fairness problem, at the optimum level.

Another solution that seems feasible is to ensure performance demands of the multiple tenants by relying on elastic scalability of the cloud system. With elasticity, the system can scale itself up or down according to the minimum necessary computational power to handle all the requests within the minimum requirements of SLA. Unfortunately, relying on elasticity does not solve the problem either. From a general perspective, both performance isolation and elasticity support the realization of SLA, but their objectives are different. The main goal of elasticity is not directly to support performance isolation but rather scalability of the SaaS in general. The system could scale up while there are still disruptive tenants who hamper the performance of other tenants.

Since complete isolation of computational resources, caching, and elasticity do not solve the identified problems at the optimal level, it is important to provide a solid approach, which achieves fairness with respect to performance requirements. We will discuss this in the following section.

8.4 PERFORMANCE MONITORING IN CLOUD-BASED SYSTEMS

In essence, to build a fair system, the following conditions must be satisfied [8,12]:

1. Customers working within their defined limits should not be affected by other customers exceeding their limits. The term *limit* here is referred to as the workload size a tenant is agreed to run on a cloud server. Workload can be considered as the amount of requests within a given time period.
2. Customers exceeding their limits and causing a negative impact of others should be reduced by performance degradation. This eventually makes the system light and responsive to each customer.
3. Customers that have better limits should have better performance compared to customers with lower limits. Performance here can be defined as input/output related parameters such as response time, request rate, etc.

To ensure performance isolation four different architectural solutions are proposed [8,12] (Fig. 8.4):

Artificial delay

This approach generates artificial delay on tenant requests by considering the tenant limitations and its corresponding request rate. The purpose behind this delay is to create backpressure to the disruptive tenant. This strategy expects increased response times for disruptive tenants. The delay time for responses can be calculated dynamically considering the limitations of the tenant, or it may be a constant value for all disruptive tenants. Experiments show that the steady amount of artificial delay cannot keep up performance isolation since it does not prioritize the disruptive requests compared to abiding requests. In this way, the same workload coming from the disruptive requests is handled by the system in a delayed manner.

Round robin

This strategy creates FIFO-queues for each tenant and gives the requests first coming out by the queue in each turn. Since these queues are handled by the request manager layer, the application server does not need to consider prioritizing the requests or tenants. The application layer simply takes the incoming request from the next request provider. The expected outcome of this technique is to distribute utilization of the system among the waiting tenants. When some queues are empty, and there are only few active tenants on the system, the empty queues are skipped and therefore the active tenants are served as much as the workload capacity of the system. Moreover, they use the workload capacity of the offline tenants, which provides a cost-optimized solution while distributing the workload.

Black list

The black list strategy uses two FIFO queues for handling the incoming requests. Typically, the first queue is used to fetch the requests and send to the application server. The second queue is the black list queue and is only served when there are no incoming requests from the first queue. Otherwise every nth request is fetched from the first queue, where n is some large number to process the blacklist queue slowly. When some tenants exceed the request limits, their requests will be redirected to the black list queue to push the distribute tenant back and process them slowly.

Thread pool

This strategy separates the workload resource pool reserved for each tenant by separating each tenants thread pools. Incoming requests first queued in the request manager layer are ordered in an FIFO basis. Subsequently, the thread pools in the application server processes the next incoming requests according to its available workload capacity.

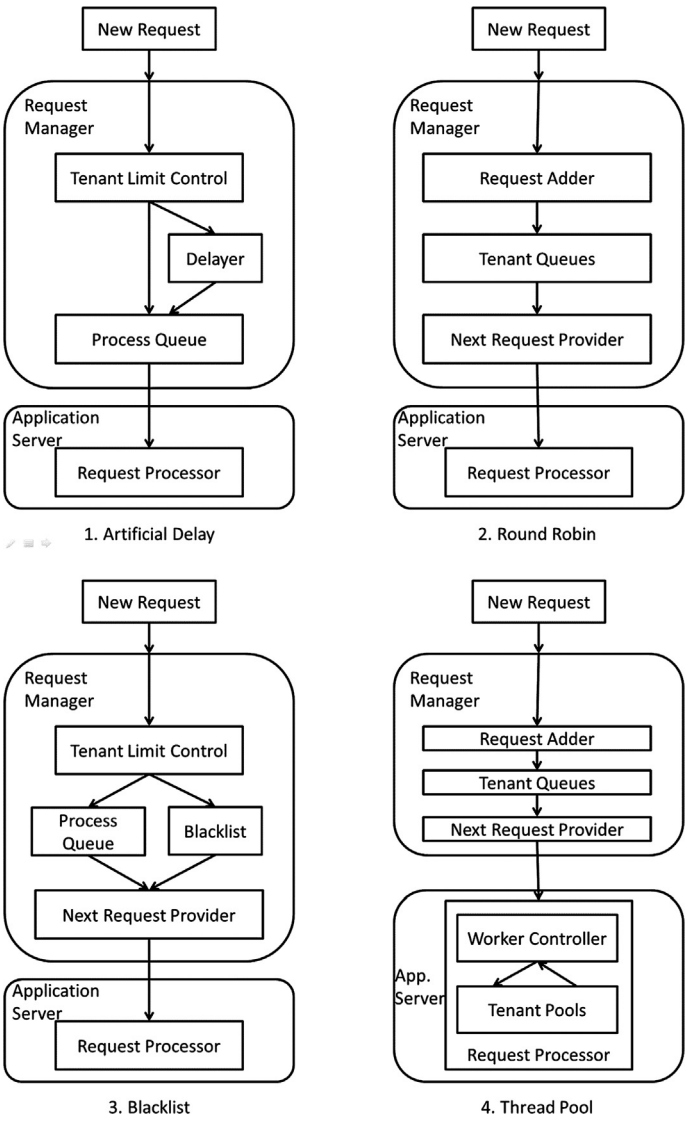


FIGURE 8.4

Proposed performance isolation approaches in Multitenant Applications (adapted from [8])

8.5 APPLICATION FRAMEWORK FOR PERFORMANCE ISOLATION

In this section, we present the so-called Tork application framework that integrates performance isolation with the existing request handling mechanisms. In the conventional case, without performance

isolation, incoming requests are first received by load balancers that on their turn redirect these requests to the most feasible application nodes. The selection of the nodes is usually based on the request traffic on the system. This approach provides a balanced request distribution for different end points and utilizes the entire system to reach the best performance. However, during the request handling, existing cloud providers do not use performance isolation strategies at the SaaS. The main reason for this is mainly to prevent unknown impact of disruptive tenants on the SaaS level business logic. In this context, request workloads are usually estimated by counting the amount of incoming requests per second (RPS), and based on this the workload is distributed among the nodes. The RPS calculation by itself, however, does not provide performance isolation and as such need to be enhanced with proper performance isolation approaches, as introduced in Section 8.4.

To meet the performance isolation in the system and calculate the overall system performance, we used the performance isolation metrics in [8,12]. To integrate the performance isolation mechanisms in the practical big data system, we assumed that the following criteria are satisfied:

1. Different entities which are sending requests to the cloud application must be identified, and separated into tenants.
2. To reach the optimal performance balance for a request endpoint, the processing times for different tenant requests should be uniformly distributed.
3. Tenant related requests need to identify themselves from the application (SaaS) layer; they should not depend on a lower layer such as TCP/IP protocol.

The first criterion is necessary to distinguish different tenants and group the same tenants in order to isolate the incoming requests from different tenants. This criterion also determines which entities in the application need isolation. Requests for each tenant may contain various amounts of workloads, which is used to prioritize the tenants.

The second criterion may not be always possible in a real world cloud application. This is because different workloads of different tenants will disrupt the assumption of a uniform distribution. A possible approach for separating the request types with different workloads is choosing the requests according to their I/O formats (such as write only requests and read only requests). Assigning similar workloads to the same endpoint will utilize the system isolation by utilizing the isolation metric predictions for the workloads.

For the third criterion, performance isolation in the SaaS layer requires tenants to identify themselves in the software layer. Therefore, all tenant related, and isolation required requests should send their tenant information in the application logic. Since the tenant information is sent from client to server, this information could be changed and may contain misleading data for the server. Handling wrong tenant data from the cloud server may cause severe performance degradations in this case, and it could create a potential security risk. To accomplish a secure cloud application, this information should be handled correctly and processed to become real tenant identifier. Therefore, during the transportation of real tenant identifier from client to server, it should be encrypted hash keys, or session based temporary keys.

The Tork Framework is designed to satisfy these three key performance isolation requirements. The conceptual architecture of a cloud-based big data system with performance isolation using the Tork framework is shown in Fig. 8.5. The Tork Framework is based on a generic client–server architecture pattern, and supports three node types including, client nodes, application nodes, and database nodes. Tork Framework uses Node.js environment in application nodes and provides JavaScript API library for the client-side development, which is the same language used in Node.js environment. The

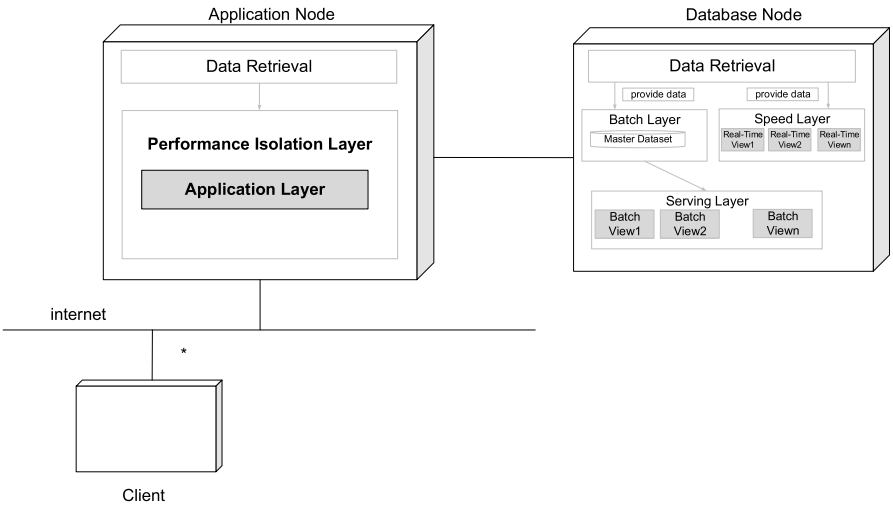


FIGURE 8.5

Performance isolation in a cloud-based big data system

framework can be used to support both the development of thin clients and thick clients. Fig. 8.5 shows an example of a thin client since the application layer is only included in the application node. Alternatively, the client node could also include the application layer. The framework supports both options.

As shown in Fig. 8.5, the Performance Isolation layer includes Application layer that represents the business logic with the performance isolation services of the SaaS application. Hereby, the business logic is based on the model-view-controller (MVC) pattern to process the requests. To realize the performance isolation, the business logic is encapsulated within the performance isolation layer. The available performance isolation algorithms are separately defined as pluggable modules and together define the overall performance isolation layer. Further, each performance isolation algorithm is configurable and can be selected and applied on the incoming business logic requests. The selected performance isolation algorithm basically intercepts the incoming requests and calculates the waiting times based on the request parameters, which is then used to determine the tenant information and the related request end point. As soon as the waiting time condition is satisfied, the intercepted request is applied to the application logic. This sequence of these actions is given in Fig. 8.6.

Besides the overall performance isolation layer, Tork Framework provides built-in common MVC Framework modules for utilizing the productivity of the business logic. Detailed information of these modules is provided in [12]. Encapsulating the usage of existing modules at the SaaS level also provides performance isolation in a more comprehensive way with respect to the business logic. These encapsulated modules may include Data-Grids, Data-Forms, Logging, Login, etc., measuring the performance isolation for unit access modules is a future research topic for the Tork Framework.

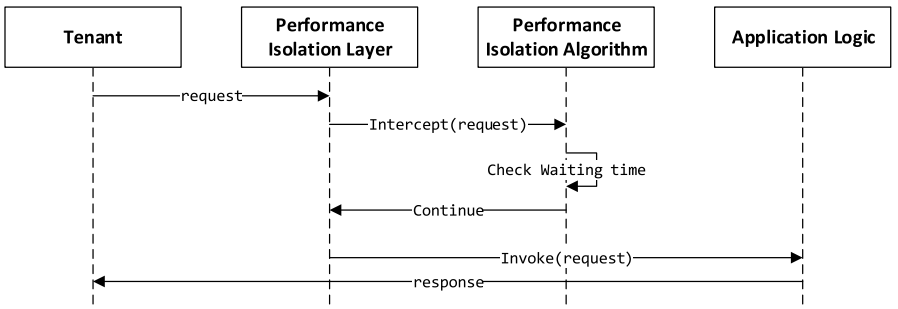


FIGURE 8.6

Sequence of requests using performance isolation

8.6 EVALUATION OF THE FRAMEWORK

To evaluate the results of performance isolation on the Tork Cloud Application Framework, we have adopted the case study of PublicFeed where the elasticity and cache mechanisms of the cloud application could not entirely solve the problem of balanced availability for each active user. To illustrate these critical points, we selected two use cases from a mobile social media application and evaluated the performance isolation algorithms under the selected cases for different scenarios.

PublicFeed provides social media services such as writing location based local news, news-feed timeline, news-feed detail for reading the published news. 10K daily active users currently use this application and their hundreds of concurrent connections operating from the single data store. Most of the functionalities are related with Feed-Timeline requests and Feed-Detail requests for all active users. These requests are served by 2 Node.js application servers for high availability and 1 database server for consistency among application servers. In the current production of the application, these requests are served from Tork Framework without using the performance isolation principles. To measure the effects of performance isolation on the system, we replicated the overall database and application nodes from the production environment, then setup a testing environment to the same cloud network. This environment included 8 vCPU and 15 GB of RAM instance for the database server and 2 vCPU and 8 GB of RAM for the application server. Microsoft SQL Database server is used for storing the overall data of the application and Node.js framework used baseline for the Tork Framework in the application layer.

Since the nature of the application requires consistent real time availability, it does not use caching in the application layers. Instead, it utilizes the database layer by calculating the changing data for each request. Changing data is based on various factors. These factors include user location, their interest of category, published feed time and general popularity of the feeds inside the overall database system. From the perspective of the data velocity that is caused by the disruptive tenants, this system can be considered a transition state of a big data system. Our unique implementation of the proposed approaches in the Tork Framework provides the performance isolation solution for the big data velocity system; therefore, we consider our practical contribution also applicable for the big data systems.

Table 8.1 Scenario parameters to be used in the evaluation framework

Scenario parameter	Description
Number of tenants	Number of active users using the application server
Number of clients	Number of active connections opened by tenants
Number of disruptive tenants	Active users that have more number of concurrent connections
Number of abiding tenants	Active users that have normal or low number of concurrent connections
Concurrent clients per disruptive tenant	Number of active connections takes service for a single active user
Concurrent clients per abiding tenant	Number of active connections takes service for a single abiding user
Service level agreement for requests	Promised request count per tenant per second
Waiting time between client requests	Calculated automatically from SLA parameter. 0 for disruptive tenants, client size/SLA for normal tenants
Adopted performance isolation approach	Selected approach for evaluating the results, it can be Round-Robin, Delay, Black-List or Nonisolated

In our use cases, we measured the isolation metrics based on two scenarios. These scenarios are the cases of low rate of requests and heavy rate of requests. In these scenarios, we used a set of parameters to determine the workload of the system. These parameters are described in Table 8.1.

To measure the isolation metrics, requests counts and response times from the real environment should be collected. To send, collect and measure the results, we have developed a web based simulation client. By using JavaScript and AJAX technologies, this client was able to open multiple concurrent HTTP connections at a time, simulate different tenants, and measure the request rate with corresponding response time for all tenants and all clients.

Since some default security settings for the modern browsers prevent high amount of TCP connections, we used the Mozilla Firefox web browser and changed its socket size limitation to use 10,000 concurrent sockets at a time, which is sufficient for this experiment.

Tenants, client amount of tenants, and their corresponding request rates are configurable from the user interface. Since we assumed that the SLA is based on acceptable request count per second, request rates are indicated as an SLA variable, which determines the usual amount of request rate; therefore, it also determines the disruptive tenants whose request rate is above the usual. These set of parameters needs to be selected before the experiment to determine the test scenario. Set of parameters are grouped into user interface as follows:

Configurations for selecting the use cases and their scenarios

For this case study, there are two use cases and two scenarios for each. These are Feed Detail and Feed Timeline use cases; low load and heavy load scenarios. In the Feed Detail use case, one of the random feed is selected first and all of the information related to this feed is requested by different tenants. Response for Feed Detail use case is a single feed, specialized with the requester tenant's unique interactions. This use case is the most basic read request from database layer. For the Feed Timeline use case, different numbers of tenants request their timeline list. Timeline list is computed at the database layer for each incoming request. Feed Timeline use case utilizes the database via complex calculations more than any other use case in the application. Low load and heavy load scenarios for each use case change the SLA variable of the use case. This affects the simulation software to send more requests per second for abiding tenants.

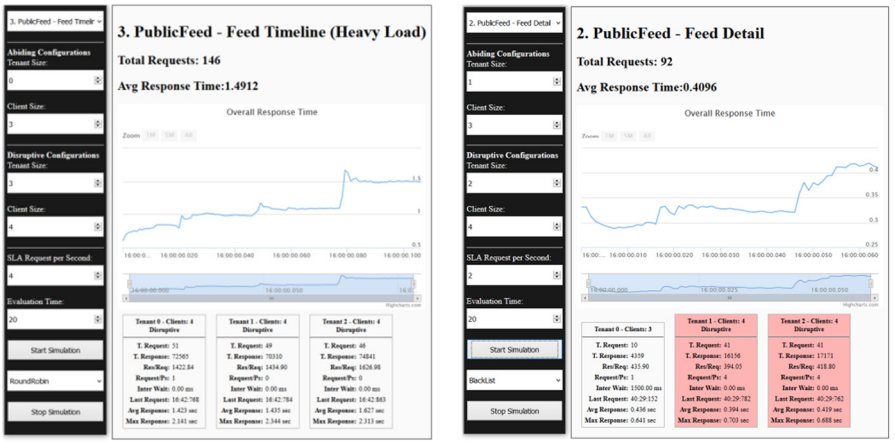


FIGURE 8.7

Different snapshots of the simulation environment

Configurations for abiding tenant and clients

Since we assumed that requirement for being abiding tenants is using the system within the limits of the SLA, the abiding tenant configuration is used to select the number of abiding active users in this case study. Related clients of these abiding tenants have a request rate within the limits of SLA. Selection of these parameters determines the number of connections that will satisfy SLA conditions. The number of connections will be equal to the tenant size multiplied by the client size.

Configurations for disruptive tenant and clients

Disruptive tenant configuration has the same parameters as the abiding tenant configuration. Disruptive connections do not have any waiting period for making another request to the server. They are constantly making requests and are assumed not to satisfy SLA.

SLA, evaluation time and algorithm configurations

Simulation software uses SLA variable as a number of requests that can be processed per second for a single tenant. When the tenants request more than SLA variable, they violate the SLA, and the related performance isolation algorithm can check this condition and may take appropriate actions. Evaluation time is a period in seconds to measure the performance isolation of the application. During this time, requests are made by clients. Their related request rates and response times are calculated and stored, and the stored data is drawn as a graph. Changing the algorithm configuration notifies the application servers and changes their way of handling requests to the selected algorithm. In addition, changing the algorithm clears the caching mechanisms of the application server that are used for request handling algorithms.

Fig. 8.7 shows two snapshots from the simulation environment during the experiments. After selection of parameters, for each use-case/scenario/algorithm pair, the tenant number of phase experiments should be done to measure the effect of disruption and isolation for the given parameters. Each phase has a duration of evaluation period variable in seconds. In each phase, abiding tenant amounts decrease by one as the disruptive tenant amounts increase by one. In each phase, simulation environment shows

Table 8.2 Evaluation metrics used for performance isolation

Symbol	Meaning
t	A tenant in the system.
D	Set of disruptive tenants exceeding their request rate limits (e.g., defined in the SLA) $\|D\| > 0$.
A	Set of abiding tenants not exceeding their request rate limits (e.g., defined in the SLA) $\|A\| > 0$.
w_t	Workload of the tenant t; $w_t \in W$, $W = \sum_{t \in AUD} w_t$.
W	Total system workload. It can be found by the addition of disruptive and abiding tenants' workloads.
$z_t(W)$	Reflects the QoS provided to tenant t, represented as real number. QoS observation of the individual tenant, which is the function of overall system workload. Lower values correspond to better qualities (low latency response time).
I	Isolation degree of the system. Index is used to separate different isolation values.

the overall response time for the requests and highlights the disruptive tenants' real time. This gives some status clue about the computation resource allocation of the application server, and may even indirectly give status clue about database server during the tests. This way, it is possible to understand whether the test parameters have a meaningful workload effect on the overall application.

8.6.1 EVALUATION RESULTS

Performance isolation is calculated by using the quality of service metrics [8,12]. The metrics that we use have adopted from [8] are shown in Table 8.2. Measuring the isolation depends on at least two other measurements. The initial measurement is the reference value and the second measurement is the disruptive case value. Reference value indicates the quality of service results for all users in the system ($t \in A$) that is indicated as W_{ref}. For the second measurement, a subset of abiding tenants challenges the system isolation by increasing their workload, this point is the disruptive case value and it is measured as W_{disr}. During the measurements of these values, the overall tenant number does not change, therefore the union of A and D remains the same.

The relative differences of these two measured values indicate the QoS of Δz_A, and this is also defined as the reference quality of service compared to the disruptive quality of service. Eq. (8.1) shows the calculation of this comparison [8]:

$$\Delta z_A = \frac{\sum_{t \in A} \left[z_t \left(W_{disr} \right) - z_t \left(W_{ref} \right) \right]}{\sum_{t \in A} z_t \left(W_{ref} \right)}. \tag{8.1}$$

Similar to the comparison of the quality of service, relative difference of the workload is measured as it is shown in Eq. (8.2):

$$\Delta W = \frac{\sum_{w_t \in W_{disr}} w_t - \sum_{w_t \in W_{ref}} w_t}{\sum_{w_t \in W_{ref}} W_t}. \tag{8.2}$$

The change in the quality of service and workloads affects the performance isolation in the following way:

$$I_{QoS} = \frac{\Delta z_A}{\Delta W}. \tag{8.3}$$

Table 8.3 Timeline-view scenario performance isolation indices

	Scenario 1: timeline view (low workload)				Scenario 1: timeline view (heavy workload)			
Approach	I_{QoS1}	I_{QoS2}	I_{QoS3}	I_{avg}	I_{QoS1}	I_{QoS2}	I_{QoS3}	I_{avg}
Non-isolated	3.69	4.05	6.14	4.62	2.56	1.58	5.48	3.10
Round robin	1.96	2.43	3.44	2.61	1.74	1.90	3.70	2.45
Delay	2.29	2.75	3.09	2.71	3.93	1.62	4.26	3.28
Black list	0.98	2.43	3.27	2.22	1.22	5.18	4.57	3.66

Table 8.4 Feed-detail scenario performance isolation indices

	Scenario 2: feed detail (low workload)				Scenario 2: feed detail (heavy workload)			
Approach	I_{QoS1}	I_{QoS2}	I_{QoS3}	I_{avg}	I_{QoS1}	I_{QoS2}	I_{QoS3}	I_{avg}
Non-isolated	1.28	1.63	2.13	1.68	1.35	1.86	2.31	1.81
Round robin	1.22	1.62	2.08	1.64	1.26	1.74	2.13	1.71
Delay	5.58	5.27	5.61	5.49	4.31	4.83	4.59	4.58
Black list	1.13	1.50	2.06	1.56	1.42	1.66	1.98	1.68

To disregard the network latency, response time metric is taken as the duration of calculation in the server, until the last byte of the response is flushed. Therefore, average response duration of $\boldsymbol{t}$ variable is calculated by a function of z_t, which results as the workload $\boldsymbol{W}$. In the experiment, workload size is derived from the request frequencies from the concurrent connections that belong to each tenant.

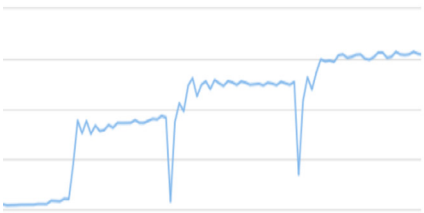
In this simulation, we considered low and heavy workload scenarios of Timeline-Request and Feed Detail Request use cases. For each case, we have examined 3 active abiding users that have 3 clients, each changing into 3 disruptive users that have 4 clients. Changing from abiding tenants to disruptive tenants in each phase, we found the related quality of service index. These quality-of-service indices are enumerated as I_{QoS1}, I_{QoS2}, and I_{QoS3}. We calculated the averaged isolation metric, I_{avg}, by averaging the quality indices.

Tables 8.3 and 8.4 show the average and steps of the service quality indices for each use case and scenario. For these indices, lower values indicate a more balanced way of workload distribution among tenants which implies better performance isolated system. For the baseline of comparison, nonisolated approach refers to the existing system whereas the others are the results of the proposed isolation algorithms.

For Table 8.3, I_{avg} shows exponential values in the nonisolated as the disruptive tenant size is increased. This means the quality of service is negatively impacted for the abiding tenants due to increase in the workload of the disruptive tenants. Therefore, this use case provides a good example to show the effects of the performance isolation algorithms as we showed the existing system is nonisolated. For the low workload scenario, Round-Robin, Delay and Black List approaches show the increasing I_{avg} as the disruptive tenants are increased. However, this increase looks constant compared to nonisolated system. The Black List approach seems to be the best approach under the low workload scenario. However, Round Robin is the best approach under heavy workload scenario. This is caused by the variation of the workload in the Time Line View scenario. In this scenario, we see that delaying or punishing the tenants due to their disruptive behavior causing worse performance isolation impacts better than the default nonisolated approaches.

Response time per request with increasing disruptive tenants in non-isolated system

Response time per request with increasing disruptive tenants in Round Robin system



Response time per request with increasing disruptive tenants in Black-List system

Response time over increasing disruptive tenants at a Delay based isolated system

FIGURE 8.8

Response time per request for different performance isolation strategies

In Table 8.4, at the Low Workload scenario we see that the system is not nonisolated at all. Since retrieving a feed detail from the server does not challenge workload, all requests are immediately served from the available computational resource. However, in this case Delay approach shows substantial increase compared to other approaches. This situation shows that delaying requests when the system has available resources causes negative impact on the performance isolation among tenants.

Algorithm – response time characteristics

So far we have analyzed the performance isolation approaches for different workloads. The performance isolation approaches are necessary to guarantee the fairness principle and as such do not violate the expected response times for each tenant. An important question here is how the overall average response time is affected with increasing disruptive tenants.

During the test case we have also investigated this issue by adopting three different phases. At each phase the number of disruptive tenants was increased and the average response time was measured. The results are shown in Fig. 8.8. The horizontal axis of the figures represents the time, while the vertical axis represents the average response time.

From Fig. 8.8 we can conclude that an increased workload from disruptive tenants actually is agnostic to the adopted performance isolation approaches. The average response time appeared to increase proportionally for all the four performance isolation approaches. For the nonisolated approach, the fairness principle will be violated but the average delay time will increase proportionally for each individual tenant. For the other performance isolation approaches, the average response time increases proportionally but fairness is not violated.

8.7 DISCUSSION

We have provided a systematic approach and a corresponding framework to integrate performance isolation within cloud-based big data systems. The study has been carefully planned and evaluated. As such, we believe that the overall approach could be of value for both practitioners and researchers. To define the scope of the work and pave the way for further study, the following issues might need further attention:

One instance is used to simulate all client requests.

For sending all client requests, the evaluation environment is used. This environment is executed on top of a single instance. This means that all of the client requests are sent from the same source. In a real life scenario, client requests may be coming from different nodes and these may belong to different geographical distances to the application server.

Small group of users and clients are used for the tests.

Since all requests can be sent from one computer, there are limiting factors coming from the testing computer. According to these limitations we used three users and four clients for each user, thus totaling 12 concurrent active connections. In a more advanced real life scenario, these numbers may be increased, and the behavior of the application server and database instances may be changed due to different utilization criteria.

DNS lookup times and ingress network traffic for the requests are not considered.

Since all requests are sent from a single computer only, the DNS lookup time is computed once for all requests. This is the first reason why the DNS lookup times are not included in the response times. The second reason is that we assumed real world scenarios for each client, there would thus be only one DNS lookup compute time; this case would not affect the behavior of the overall cloud system. However, in some more advanced real life scenarios, client behavior may vary and active clients may tend to be unique clients that could be from different geographical areas. In this situation, since they connect to the application server at the first time, the DNS lookup time and ingress network traffic may also effect to response time, which may change the results.

Application server is artificially limited for test scenarios.

Since we have limited computational power to simulate all clients, and wanted to examine the performance isolation solutions under heavy utilization, we limited the application server. This limitation is based on CPU throttling of the application layer, and performance isolation layer processes. Therefore, only a limited amount of hardware is used to work for processing the requests. To settle this limitation, first we tried different values of CPU percentages, and selected the best value according to the utilization of the response time. When the response time was increased, and stabilized, responding to all the client requests, we understood that limited hardware amount is utilized for the test cases. However, in general, finding the system limitation given client requests is not a feasible thing to calculate because of the varying workloads and different operating system limitations.

8.8 RELATED WORK

Several cloud application development frameworks have been proposed in the literature. For example, Intercloud [2], Appscale [4], CloudScale [14] and EasySaaS [19] are recent popular frameworks and publications that are focused on scaling multitenant cloud applications. However, none of these

explicitly consider performance isolation. Moreover, they primarily provide elasticity solutions for supporting QoS of the cloud application. As stated before, elasticity is important for enhancing performance but it does not directly solve the problem of fairness in case of disruptive tenants.

Performance isolation has been addressed in the literature. Hereby, the focus has been generally on resource efficiency, and performance isolation metrics on multitenant architectures. It is based on resource sharing by optimal placement of customers given a set of resources, and SLA requirements [6,21]. Optimal placement solutions enhance the performance isolation in the application; however, it cannot be considered as an alternative for performance isolation.

Schroeter et al. [13] provide a tenant-aware model that can be reconfigured automatically. This leverages the isolation by placing disruptive tenants into one single node or adding a new node to the system. However, the contribution of the paper is based on elasticity and not directly on performance isolation as we have proposed in this paper.

Lin et al. show an approach to achieve performance isolation on multitenant applications [9]. Hereby, different quality of service is provided to different tenants. The approaches are evaluated based on a test case with changing workloads. However, this approach mainly focuses on differentiating QoS, and does not directly relate to supporting performance isolation.

Wei et al. [20] proposes a technique for isolation of resources by estimating the demands of tenants. Performance isolation is achieved by preventing the disruptive tenants that are responsible for the degradation of the performance on other tenants. The approach is based on static control of resources for the performance isolation and does not use dynamic adaptation of the resources. Further, the approach does not directly consider service level agreement (SLAs), since controlling resources does not directly guarantee fulfilling the SLAs [5].

Performance isolation is actually one way to deal with the problem of performance interference. In [1,3] and [11], instead of admission control and/or isolation, the authors formulate the performance interference problem as an optimization/control problem. Subsequently, they try to mitigate interference by making optimal and the fairest trade-off for all tenants in cloud. In this case for realizing fairness primarily elasticity is adopted. We consider this complementary to the performance isolation approach.

8.9 CONCLUSION

Performance monitoring in cloud-based big data systems is an important challenge that has not been fully solved, yet. We have identified and discussed several potential solutions including caching and scalability. However, none of these approaches completely solves the problem of disruptive tenants that exceed their performance resources and as such impede the performance of other tenants. Considering the fairness criteria, we have discussed that performance isolation is an important concern. Several performance isolation strategies have been identified in the literature. We have provided an approach and discussed the application framework Tork that can be used to integrate performance isolation mechanisms in existing cloud-based big data systems. The framework has been applied to PublicFeed, a social media application that is built on top of a cloud-based big data system. We have reported on the experimental simulations which consider the application of three different performance isolation strategies. Depending on the simulations, we conclude that adopting performance isolation strategies clearly enhances the performance of cloud-based big data systems. Interestingly, the problems that are difficult to solve using the conventional caching and scalability approaches can be supported using

performance isolation. In our future work, we will further experiment and simulate with different industrial case studies. Hereby, we will also consider the combination of different performance management approaches of caching, scalability, and performance isolation.

REFERENCES

[1] X. Bu, J. Rao, C. Zhong Xu, Coordinated self-configuration of virtual machines and appliances using a model-free learning approach, IEEE Trans. Parallel Distrib. Syst. 24 (4) (Apr. 2013) 681–690.

[2] R. Buyya, R. Ranjan, R.N. Calheiros, Intercloud: utility-oriented federation of cloud computing environments for scaling of application services, in: Algorithms and Architectures for Parallel Processing, in: LNCS, 2010, pp. 13–31.

[3] T. Chen, R. Bahsoon, Self-adaptive trade-off decision making for autoscaling cloud-based services, IEEE Trans. Serv. Comput. (Nov. 2015).

[4] N. Chohan, et al., Appscale: scalable and open AppEngine application development and deployment, in: Cloud Computing, Springer, Berlin, Heidelberg, 2010, pp. 57–70.

[5] V.C. Emeakaroha, et al., Low level metrics to high level SLAs-LoM2HiS framework: bridging the gap between monitored metrics and SLA parameters in cloud environments, in: 2010 International Conference on High Performance Computing and Simulation (HPCS), IEEE, 2010.

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

[7] D. Jacobs, S. Aulbach, Ruminations on multi-tenant databases, BTW 103 (2007).

[8] R. Krebs, C. Momm, S. Kounev, Metrics and techniques for quantifying performance isolation in cloud environments, Sci. Comput. Program. 90 (2014) 116–134.

[9] H. Lin, K. Sun, S. Zhao, Y. Han, Feedback-control-based performance regulation for multi-tenant applications, in: 2009 15th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2009, pp. 134–141.

[10] N. Marz, J. Warren, Big Data: Principles and Best Practices of Scalable Realtime Data Systems, Manning Publications Co., 2015.

[11] R. Nathuji, A. Kansal, A. Ghaffarkhah, Q-clouds: managing performance interference effects for QoS-aware clouds, in: Proceedings of the 5th European Conference on Computer Systems, ACM, 2010.

[12] O.A. Oral, B. Tekinerdogan, Supporting performance isolation in software as a service systems with rich clients, in: 2015 IEEE International Congress on Big Data (BigData Congress), IEEE, 2015.

[13] J. Schroeter, S. Cech, S. Götz, C. Wilke, U. Aßmann, Towards modeling a variable architecture for multi-tenant SaaS-applications, in: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, ACM, 2012.

[14] Z. Shen, S. Subbiah, X. Gu, J. Wilkes, CloudScale: elastic resource scaling for multi-tenant cloud systems, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, ACM, 2011, p. 5.

[15] K. Öztürk, B. Tekinerdogan, Feature modeling of software as a service domain to support application architecture design, in: Proc. of the Sixth International Conference on Software Engineering Advances (ICSEA 2011), Barcelona, Spain, Oct. 2011.

[16] B. Tekinerdogan, K. Öztürk, Feature-driven design of SaaS architectures, in: Z. Mahmood, S. Saeed (Eds.), Software Engineering Frameworks for Cloud Computing Paradigm, Springer-Verlag, London, 2013, pp. 189–212.

[17] B. Tekinerdogan, K. Öztürk, A. Dogru, Modeling and reasoning about design alternatives of software as a service architectures, in: Proc. Architecting Cloud Computing Applications and Systems Workshop, 9th Working IEEE/IFIP Conference on Software Architecture, 20–24 June 2011, pp. 312–319.

[18] B. Tekinerdogan, Software architecture, in: T. Gonzalez, J.L. Díaz-Herrera (Eds.), Computer Science Handbook, Volume I: Computer Science and Software Engineering, second edition, Taylor and Francis, 2014.

[19] W.T. Tsai, Y. Huang, Q. Shao, EasySaaS: a SaaS development framework, in: 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA), IEEE, 2011.

[20] W. Wang, et al., Application-level CPU consumption estimation: towards performance isolation of multi-tenancy web applications, in: 2012 IEEE 5th International Conference on Cloud Computing (Cloud), IEEE, 2012.

[21] Y. Zhang, Z. Wang, B. Gao, C. Guo, W. Sun, X. Li, An effective heuristic for on-line tenant placement problem in SaaS, in: 2010 IEEE International Conference on Web Services (ICWS), IEEE, 2010.