2014 IEEE 8th International Symposium on Service Oriented System Engineering

A QoS-aware Load Balancing Policy in Multi-tenancy Environment

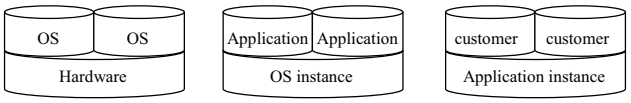
Hailong Sun, Tao Zhao, Yu Tang, Xudong Liu
School of Computer Science and Engineering
Beihang University
Beijing,China
Email: {$sunhl, zhaotao, tangyu, liuxd$}@act.buaa.edu.cn

***Abstract*—Cloud computing aims at providing services on the basis of a shared pool of underpinning resources and load balancing is of paramount importance in such an environment. At the same time, multi-tenancy is widely adopted in cloud computing to reduce the costs of service provisioning and to improve resource utilization. Multi-tenancy brings new challenges to load balancing, since it incurs resource competition and different QoS requirements of hosted applications. Therefore, servers with multiple deployed applications need a proper request scheduling policy to guarantee their quality of service, e.g., response time. However, most of the QoS-aware load balancing algorithms do not concern about the mutual intervention among applications deployed on the same server. When under heavy loads, mean response time of some applications may become too high to be acceptable. In this work, we propose a new load balancing algorithm, "Server Throughput Restriction(STR)", based on M/G/s/s+r queueing model, in order to guarantee each application's mean response time and also achieve better server throughput. In addition, we conduct several experiments to analyze the performance of STR in comparison with Round-Robin and Least-Work-Remaining.**

I. Introduction

Cloud computing is emerging as a new paradigm of large-scale distributed computing that enables convenient and on-demand network access to a shared pool of configurable computing resources [1] [2]. Both the industry and the academia have witnessed its rapid rise in recent years, but it still remains far from being mature as there are many existing challenging issues that have not been well addressed. Load balancing is one of the important issues in cloud computing, which is responsible for distributing the workload evenly to processing nodes so as to meet users' QoS(Quality of Service) requirements and to reduce the costs of service providers. It helps in avoiding overload among system components and enabling scalability in the cloud, thus improves the resource utility and system performance.

However, load balancing in cloud computing differs from that in traditional environments like cluster or grid computing due to the sharing of resources with multi-tenancy, which enables the isolation among tenants and lowers the cost for service providers. As a matter of fact, multi-tenancy has different meanings in different service layers of the cloud computing model, as illustrated in Figure 1. In the IaaS layer, multi-tenancy means that multiple operating system instances, usually in the form of virtual machines, share the same physical hardware through a hypervisor. In the context of PaaS layer, multi-tenancy means that different applications share the same operating system instance. SaaS multi-tenancy, which has the highest level of isolation, means that one application instance is shared across multiple customers [3].



(a) IaaS multi-tenancy (b) PaaS multi-tenancy (c) SaaS multi-tenancy

Fig. 1: Three kinds of multi-tenancy

Nowadays, PaaS providers such as Heroku [4] and CloudBees [5] isolate code from different applications on the same OS instance so as to improve resource utilization, which brings new challenge to load balancing as the server is shared by several applications. Most of the QoS-aware load balancing algorithms [6] [7] [8] [9] are not suitable in this context since they do not concern about the mutual intervention among applications on the same server. This way, if the applications have some kind of QoS demands such as that the mean response time would not be higher than an acceptable threshold, some of the applications' response time may not be guaranteed due to the competition for shared resources. Therefore, in this work, we mainly focus on QoS-aware load balancing issues for multi-tenancy application environments in PaaS layer, and specifically we aim at ensuring that each application's mean response time should not be higher than its corresponding threshold in spite of resource competing among applications.

In this work, a new load balancing policy named STR is proposed for multi-tenancy environment with CPU-intensive applications such as that process videos or images uploaded by users. We assume that applications' request time distribution on an idle server is known a priori, then we use M/G/s/s+r queueing model to analyze the mean response time of each application and obtain the throughput restriction equation of each server based on the concept of traffic intensity. Therefore, requests are dispatched to servers that meet throughput restriction. In addition, STR has been implemented on the basis of our Service4All platform [10], which is a multi-tenancy PaaS platform that supports application hosting and development with different programming models.

The main contribution of this work includes:

- We design a novel method to calculate the throughput restriction on each server according to the mean re-

978-1-4799-3616-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SOSE.2014.21

sponse time demands of deployed applications based on M/G/s/s+r queueing model, which makes no assumption about the idle request time distribution of applications, such as exponential or heavy-tailed.

- We propose a new load balancing algorithm in PaaS multi-tenancy environment, which guarantees each application's mean response time and also achieves better server throughput.
- We build a load balancer that implements our algorithm on the basis of our Service4All platform, and conduct an extensive set of experiments to compare the performance of the algorithms in throughput and mean response time based on a multi-tenancy server cluster.

The rest of the paper is organized as follows. Section II briefly introduces the background of queueing theory and the performance measures of M/G/s/s+r queue. In Section III, we describe the calculation of throughput restriction on a server. Section IV discusses the load balancing algorithms in detail. In Section V, we provide the experiment process and results, as well as our analysis. Section VI introduces the related work of load balancing polices. Finally, we come to the conclusion and discussion of the future work in Section VII.

II. Preliminaries

Intuitively, the response time of an application is not only dependent on the processing speed of computing nodes, but also affected by the request arrival rate. When under heavy loads, requests are put to waiting in queues due to limited CPU resources, which will lead to higher response time. Therefore, we can use queueing theory to analyze the request processing in a server. In this section, we first briefly introduce the background of queueing theory, then give the performance measures of M/G/s/s+r queue.

A. Queueing Theory

Queueing theory is the mathematical study of waiting lines, or queues [11]. As is shown in Figure 2, a queueing system mainly consists of three fundamental parts: task arriving process that specifies the input rule of the system, queueing discipline that represents the waiting rule of tasks, and servicing process that indicates the output rule of the system. Then the response time of a task is the sum of its waiting time and processing time.

In the theory of stochastic process, a queueing system can be transformed to a markov chain with states that represent the number of tasks in the system at a particular moment, including tasks in the queue and in servers. The state transition is triggered by the arrival and completion of tasks. Therefore, once the arrival process and the servicing process is known, the state transition matrix of a queueing system can be constructed. Many theorems in queueing theory use the transition matrix to calculate the stationary distribution of the states and further obtain the performance measures of the queueing system, such as the mean queue length or the average response time of tasks.

As an extension to Kendall's notation [12], a queueing model can be described by six symbols: A/S/c/K/N/D, the meanings of which are as follows:

Fig. 2: A queueing system

- A: Interval distribution between arrivals.
- S: Service time distribution.
- c: Number of processing servers.
- K: The maximum number of tasks the system can hold.
- N: Size of the population of tasks.
- D: Queueing discipline.

When the last two symbols are not specified, it is assumed that there are infinite number of tasks and the queueing discipline is FCFS.

B. M/G/s/s+r Model

As stated in the beginning of this section, a server can be abstracted as a queueing system, and a suitable queueing model is needed. To begin with, the interval distribution between arrivals is exponential as requests usually come in Poisson process. The service time distribution is general, because applications' time distribution is usually indeterminate and unequal, including but not limited to exponential and heavy-tailed [13]. The number of CPUs in a server is the number of processing servers in the model, since requests are CPU-intensive. Finally, there are finite waiting rooms in the model due to limited request queue in the server. Therefore, we choose M/G/s/s+r model as the most appropriate. However, the memoryless property of the exponential distribution no longer obtains because of the general service time distribution, thus closed-form solution for the M/G/s/s+r model is difficult to construct [14]. In this paper, we use T.Kimura's approximation approach [15] that gives the steady-state distribution of the number of tasks in closed form:

$$P_j = \begin{cases} \frac{(s\rho)^j}{j!}P_0, & \text{j = 0,1,...,s-1} \\ \frac{(s\rho)^s}{s!}\frac{1-\delta}{1-\rho}\delta^{j-s}P_0, & \text{j = s,s+1,...,s+r-1} \\ \frac{(s\rho)^s}{s!}\delta^r P_0 & \text{j = s+r} \end{cases} \quad (1)$$

with

$$P_0 = \left[\sum_{j=0}^{s-1}\frac{(s\rho)^j}{j!} + \frac{(s\rho)^s}{s!}\frac{1-\rho\delta^r}{1-\rho}\right]^{-1}$$

$$\delta = \frac{\rho R_G}{1-\rho+\rho R_G}, \rho = \frac{\lambda\mu}{s} < 1$$

where R_G is the ratio of the mean waiting time in the M/G/s and M/M/s queues. A common useful approximation for R_G is given by Lee and Longton [16]:

$$R_G = \frac{EW(M/G/s)}{EW(M/M/s)} = \frac{1+C_v^2}{2}$$

TABLE I: The definition of symbols

Symbol	Definition
s	Number of CPUs
r	Request queue length
λ	Poisson arrival rate of requests
μ	Expectation of the applications' time distribution, and the mean service rate is $1/\mu$
C_v	Coefficient of variation of the applications' time distribution
ρ	Traffic intensity
P_0	Unconditional probability that there is no request in the server
P_{s+r}	Blocking probability that the request queue is full
L	Average number of requests in the server
W	Average response time of requests

Table I shows the meaning of symbols used in this section. By combining Equation 1 and the Little's law(notice that the actual arrival rate is no longer λ due to the blocking of requests), the mean response time in M/G/s/s+r model can be derived by:

$$W = \frac{L}{\lambda(1-P_{s+r})} = \frac{\sum_{j=0}^{s+r} jP_j}{\lambda(1-P_{s+r})} \tag{2}$$

Although Equation 2 gives the closed form solution of W, it is unlikely to be applied directly due to its complexity. However, W can be further simplified by reducing insignificant parameters. We found that the queue length has little influence on W since r is large enough in practice. In addition, W is positively correlated with r, thus we can safely analyze the upper bound of W, i.e. the limit of W of r, as r approaches infinity. The simplified expression of the mean response time has the form:

$$\begin{aligned} W^{'} &= \lim_{r\to\infty} W \\ &= \frac{\frac{s!}{(s\rho)^{s-1}}\sum_{j=0}^{s-1}\frac{(s\rho)^j}{j!} - s + \frac{s-s\delta+\delta}{(1-\rho)(1-\delta)}}{\lambda\left(\frac{s!}{(s\rho)^s}\sum_{j=0}^{s-1}\frac{(s\rho)^j}{j!} + \frac{1}{1-\rho}\right)} \\ &= \mu + \frac{\mu R_G}{s\left(\frac{s!}{(\lambda\mu)^s}\sum_{j=0}^{s-1}\frac{(\lambda\mu)^j}{j!}\left(1-\frac{\lambda\mu}{s}\right)^2 + 1 - \frac{\lambda\mu}{s}\right)} \\ &= \mu + \Delta(\lambda) \end{aligned} \tag{3}$$

For the request that has been dispatched to a server, it is likely to be refused when the queue length is not large enough. We need to compute each server's queue length to lower the probability that blocks requests. Equation 4 gives the relationship between the queue length and the blocking probability. Therefore, we can set up the length of the request queue to guarantee that the blocking probability belows a reasonable value.

$$P_{s+r} = \frac{\delta^r}{\frac{s!}{(s\rho)^s}\sum_{j=0}^{s-1}\frac{(s\rho)^j}{j!} + \frac{1-\rho\delta^r}{1-\rho}} \tag{4}$$

III. Throughput Restriction Calculation

From Equation 3 we can infer that given the application's time distribution and the number of CPUs, the mean response time of the application is positively correlated with the arrival rate of requests. In other words, once we restrict the maximum throughput that a server could handle, the average response time of the deployed applications should be controlled as well. In this section, we first discuss throughput restriction calculation under single-application circumstances, then we will extend it to multi-tenancy environment.

A. Single-Application Environment

For a server that has one deployed application, the maximum throughput of the application λ_{max} can be obtained by solving the following optimization equation:

$$\max_{\lambda} W^{'} \leq T_R \tag{5}$$

where T_R is the threshold of the application's mean response time. When the actual arrival rate is less than λ_{max}, all the requests would be dispatched to the hosting server and the mean response time is guaranteed. On the other hand, when the actual arrival rate is greater, extra requests should be dispatched to other hosting servers that do not exceed its corresponding throughput restriction. For the situation that no servers meet restriction, the service provider can start a new server or simply discard the following requests, which is beyond the scope of this paper.

B. Multi-tenancy Environment

In the context of multi-tenancy, a server may have several deployed applications [4] [5] and the maximum throughput is needed so that each application's mean response time is guaranteed. It is difficult to solve directly since applications do not have the same threshold. To begin with, consider the single-application situation that the maximum throughput λ_{max} can be easily obtained by the unique threshold. Next, λ_{max} can be transformed to the maximum traffic intensity by its definition $\rho = \lambda\mu/s$, which is the maximum intensity that each of the server's CPU could reach(Although $\rho < 1$, it's not the CPU utilization). Thus each application's mean response time is also positively correlated with ρ. For the applications deployed on the same server, each of their throughput restriction and corresponding traffic intensity in single-application environment can be calculated, then we choose the minimum traffic intensity $\rho_{min} = \min_i \rho_i$ on the assumption that the load of application with minimum traffic intensity is not empty, where i is the index of the applications on a server, and ρ_i is the calculated maximum traffic intensity in single-application environment for application i. We believe that when the multi-tenancy server's CPUs have the traffic intensity ρ_{min}, all the deployed applications' response time would be guaranteed.

Let λ_i be the arrival rate of requests. All applications can be seen as an integrated one with the arrival rate of $\sum \lambda_i$ and the mean request time of $\sum \lambda_i\mu_i / \sum \lambda_i$. Then the following equation can be obtained:

$$\sum \lambda_i \cdot \frac{\sum \lambda_i\mu_i}{\sum \lambda_i} = \sum \lambda_i\mu_i \leq s \min_i \rho_i \tag{6}$$

The right side of Equation 6 specifies the server's total throughput restriction, while the other side is the linear combination of each application's request arrival rate with weights that represent the expectation of the time distribution.

When the applications' loads are small or staggered, the linear combination of the arrival rates would satisfy the inequality. Therefore, they can be deployed on the same server with guaranteed mean response time, thus improves resource utilization and flexibility. We call Equation 6 the server's throughput restriction equation.

IV. Load Balancing Policy

This section proposes our Server Throughput Restriction based load balancing policy, or STR in short, which is mainly composed of two concurrent phases: the calculation phase and the scheduling phase. The calculation phase is responsible for data collecting that includes the deployment of applications and their QoS demands, as well as throughput restriction calculating that updates each server's throughput restriction when its corresponding information has changed. The load balancer is always executing the scheduling phase that dispatches requests to the server that meets throughput restriction. STR is a time slot based strategy with available throughput reset to the throughput restriction at the beginning of each slot. Therefore, the throughput restriction in STR is equivalent to the limitation of number of requests with different weights in a time slot.

For each incoming request, STR will first extract the application context that the request belongs to from its HTTP head, according to which STR is able to look up the request weight and the server list that the application is deployed from local configuration file, then dispatch the request to the server that meet throughput restriction and update the server's available throughput. If no server is available, the request is rejected by the load balancer as its admittance would lead to performance deterioration of some applications. The following part shows the detailed description of the scheduling phase of STR policy:

Algorithm 1: Load balancing with STR

```
1:  for each request do
2:     context ← extractApplicationContext(request)
3:     server_list ← context.server_list
4:     weight ← context.weight
5:     server ← maxAvailableThroughput(server_list)
6:     if weight > server.available_throughput then
7:        NOT_ADMITT(request)
8:     else
9:        update(server.available_throughput,weight)
10:       send(request,server)
11:    end if
12: end for
```

We will compare our work with two traditional and commonly used load balancing policies. One is Round Robin(RR), which is a static strategy that simply dispatches requests to backend servers alternately. The other is Least Work Remaining(LWR) that dispatches requests to the server with minimum CPU load, thus it's a dynamic strategy. However, LWR in multi-tenancy environment differs from that in single-application. First of all, LWR is also a time slot based strategy because there would be too much traffic if queries for CPU load per request. Secondly, requests during a slot cannot be dispatched to the server with minimum CPU load, otherwise the chosen server may incur burst load since all requests of the applications deployed on that server are put together. Instead, we use the idle CPU utilization based weighted round robin during each slot, which is more reasonable because all servers are involved and servers with lower CPU load get more requests to serve.

V. Experiment

We use the Service4All platform to establish a server cluster that includes five servers and clients as well as a load balancer. The server side uses Apache Tomcat running on the system of Windows Server, and the request queue length is set up to 1000 so that the blocking probability is small enough(less than 10^{-7}). The client side uses our own tool to generate requests in Poisson process. The load balancer implements three load balancing policies: STR, RR and LWR. This section first examines the single-application throughput restriction in a real server so as to regulate the computed values in Section III. Next, we compare the three load balancing policies in multi-tenancy environment under the same workload and give the graphical results.

A. Throughput Restriction

Although the throughput restriction in single-application environment can be computed by Equation 5, errors may exist in the real case: the actual value is usually lower than the calculated one. This section mainly measures the actual throughput restriction in a server and analyzes its relationship with the calculated value. Then we can apply this relationship to make the calculated values more accurate in the load balancing experiment.

We constructed several applications with different mean CPU time, and they all subject to uniform distribution and have the same response time threshold. Specific CPU time can be reached by looping enough time in the kernel mode and the user mode through Windows API. Therefore, applications in different machines have the same mean CPU time despite their differences in performance. We individually deploy the applications and separately measure the maximum throughput to keep the mean response time below the threshold in the dual-core and quad-core physical machines. Table II shows the calculated and the actual values of throughput restriction and traffic intensity.

The mean CPU time is the integer times of 15.6 because the refresh time interval of thread times in Windows is 15.6ms. We first computed the maximum throughput and traffic intensity by Equation 5, rows 4-7 in Table II show the calculated values in dual-core and quad-core machines.

The measurement for maximum throughput is conducted by gradually changing the value of the Poisson arrival rate and finishes when the gap between mean response time and threshold is less than 1ms or the step is equal to 0.01, which is the change size of request arrival rate. Given an initial value of step(which is set to 2.56), the request arrival rate will increase(decrease) when the mean response time is less(greater) than the threshold. The value of step is not halved until the request arrival rate begins to decrease(increase), and is halved after every change of the request arrival rate since

TABLE II: Throughput restriction results

mean CPU time		31.2ms	93.6ms	156ms	218.4ms	280.8ms
threshold		500ms				
calculated values						
dual-core	λ_{max}	63.06/s	20.23/s	11.58/s	7.77/s	5.56/s
	ρ_{max}	0.98377	0.94696	0.90287	0.84887	0.78075
quad-core	λ_{max}	127.16/s	41.58/s	24.35/s	16.84/s	12.54/s
	ρ_{max}	0.99183	0.97292	0.94953	0.91972	0.8802
actual values						
dual-core	λ_{max}	58.46/s	18.68/s	10.42/s	6.86/s	4.67/s
	ρ_{max}	0.91198	0.87422	0.81276	0.74911	0.65567
	error	7.3%	7.7%	10%	11.8%	16%
quad-core	λ_{max}	124.36/s	40.38/s	22.98/s	15.65/s	11.37/s
	ρ_{max}	0.97001	0.94489	0.89622	0.85449	0.79817
	error	2.2%	2.9%	5.6%	7.1%	9.3%

TABLE III: Fitting results

	fitting equation	R-square	Adjusted R-square	RMSE
dual-core	y=-43.46*x + 49.36	0.9697	0.9595	0.713
quad-core	y=-65.86*x + 67.52	0.978	0.9707	0.5035

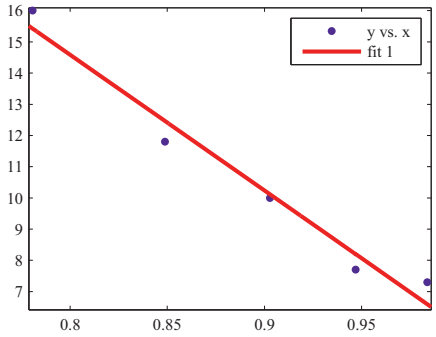


Fig. 3: Linear fitting in dual-core

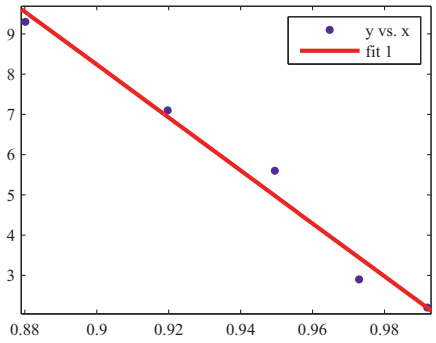


Fig. 4: Linear fitting in quad-core

then. Rows 9-14 in Table II show the actual values and errors in dual-core and quad-core machines.

From Table II we can infer that the actual values are smaller than the calculated ones, but they all follow the same trend, which means that the traffic intensity decreases as the mean

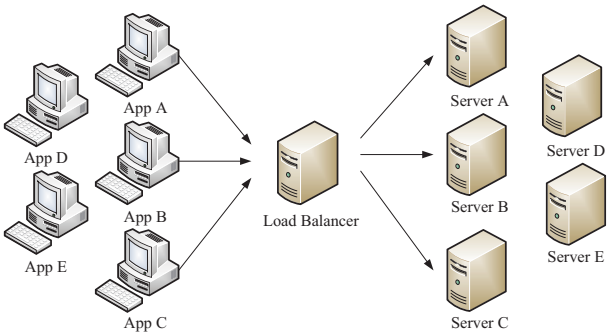


Fig. 5: Server cluster set up in our experiment

CPU time closes to the threshold. However, the actual values decrease faster, which results in greater errors. In addition, the traffic intensity in quad-core has greater values, but the errors are smaller in comparison with dual-core. Errors may exist in two ways, one is the experimental error and the other comes from Equation 5. We believe the latter is the main source for errors as Equation 5 is an approximation solution. However, errors can be represented by curve fitting method with another appropriate variable that contains all necessary information(including the mean CPU time and the threshold), which is chosen as the calculated traffic intensity.

Table III shows the linear fitting results for errors, where x stands for the calculated traffic intensity and y stands for the errors, which have different forms in dual-core and quad-core, but they all have a better goodness of fit. Figure 3 and 4 shows the graphical results of linear fitting in dual-core and quad-core. Therefore, we can use the fitting equation to eliminate the errors and make the calculated values more close to the actual ones.

B. Load Balance

In this section, three load balancing policies are compared under the same heavy workload, which is generated by five clients for each of the applications, as shown in Figure 5. The server side is multi-tenancy environment that each server has several deployed applications. We first check whether the load balancing strategies could guarantee each application's mean response time, then compare each strategy's distribution of workload on each server. A fine load balancing policy should guarantee every application's mean response time and make the allocation of request loads as fair as possible. The throughput on a server during a time slot(which is set up to 1 second) is not the sum of request number of each deployed application, but the linear combination of number of requests with different weights. To eliminate the transient phase, we discard the results obtained in the first 30 minutes, and only record data in the next 4 hours.

1) Experiment Setup: As is shown in Table IV, we construct five applications with different time distribution and mean response time thresholds. μ stands for the mean CPU time, C_v is the coefficient of variation, which is the dispersion of the time distribution. Therefore, C_v equals zero means the uniform distribution. We also give the maximum traffic

TABLE IV: Application setup

	time distribution	threshold	ρ'(dual-core)	ρ'(quad-core)
APP A	μ=31.2ms,C_v=0	500ms	0.91878	0.97003
APP B	μ=62.4ms,C_v=0	600ms	0.90308	0.96045
APP C	μ=62.4ms,C_v=1.5	600ms	0.83012	0.91439
APP D	μ=31.2ms,C_v=2	500ms	0.8415	0.92174
APP E	μ=93.6ms,C_v=0	700ms	0.89138	0.95324

TABLE V: Server setup

	CPU cores	deployment	throughput restriction equation
Server A	4	APP A, C, E	$\lambda_A^1 + 2\lambda_C^1 + 3\lambda_E^1 \leq 117.23$
Server B	2	APP B, C, E	$2\lambda_B^1 + 2\lambda_C^2 + 3\lambda_E^2 \leq 53.21$
Server C	2	APP C, D	$2\lambda_C^3 + \lambda_D^1 \leq 53.21$
Server D	2	APP A, D, E	$\lambda_A^2 + \lambda_D^2 + 3\lambda_E^3 \leq 53.94$
Server E	4	APP B, D, E	$2\lambda_B^2 + \lambda_D^3 + 3\lambda_E^4 \leq 118.17$

intensity ρ' in dual-core and quad-core, which has been revised by the fitting equations.

Table V shows the detailed setup on servers. The server side contains three dual-core and two quad-core machines with the same 2GB RAM, which is sufficient for our experiment as the applications are CPU-intensive. The deployment is chosen as all applications are deployed on servers with different CPU cores. We also give the throughput restriction equation of each server by Equation 6. λ_j^i means the request rate of the ith replica of APP j. Thus the web cluster's maximum throughput during each time slot is 395.76, which is the sum of the right side of the equations in Table V.

The setup on the client side is: λ_A=λ_D=60, λ_B=λ_C=50, λ_E=25, where λ_i represents the Poisson arrival rate of APP i. Thus the total generated load during each time slot is $\sum \omega_i \lambda_i = 395$($\omega_i$ is the request weight of APP i, as shown in Table V), which is a heavy workload since it is very close to the maximum throughput of the web cluster.

2) Obtained Results: We mainly analyze the response time of each application and the actual throughput on each server, as well as the cluster's total throughput when applying the three load balancing algorithms. Experiment results show that when under heavy loads, STR is better than other approaches in all respects, which turns out to be an effective load balancing strategy in multi-tenancy environment.

Figure 7 shows the response time results of each application. The straight dashed line in each graph specifies the corresponding response time threshold. It can be noticed that each application's response times are below the threshold line at all times when applying the STR policy, which proves that STR is able to guarantee all the applications' response times. However, RR strategy performs worse as the response times exceed the threshold at all times for each application. LWR policy has a better performance in comparison with RR, but it could only partly guarantee the response times of APP A and APP D(see Figure 7a and 7d) while other applications all exceed their response time thresholds. In addition, STR has the lowest response time values, next is LWR, and RR has the highest values.

The response time results mentioned above can be explained by the actual throughput on each server, which is shown in Figure 8. This time the straight dashed line specifies the throughput restriction of each server. STR is able to guarantee the response times because it will first guarantee that the actual throughput on each server doesn't exceed its corresponding restriction. As to the other approaches, Figure 8b and 8d show that RR strategy makes the actual throughput on Server B and Server D exceed their restriction, thus the deployed applications(APP A-E) all have higher response times despite the fact that other servers hosting their replicas may not exceed the throughput restriction. On the other hand, LWR makes Server B exceed its restriction, which results in higher response time of APP B, APP C, and APP E. In addition, Figure 8c and 8d illustrate that the actual throughput on Server C and Server D under LWR partly exceeds their restrictions, thus makes the response time of APP A and APP D close to their thresholds. When considering the distribution of workloads, STR also has a better performance as the loads are fairly allocated among the servers: the actual throughput on each server is very close to its corresponding restriction. LWR performs better than RR since the former is aware of the actual loads on each server, but the workload is still unbalanced according to Figure 8b and 8e.

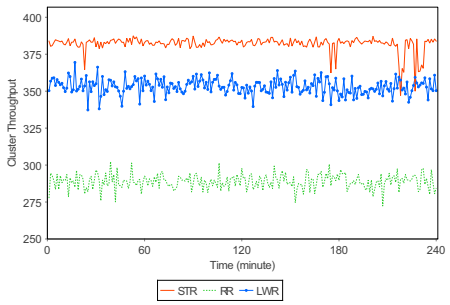
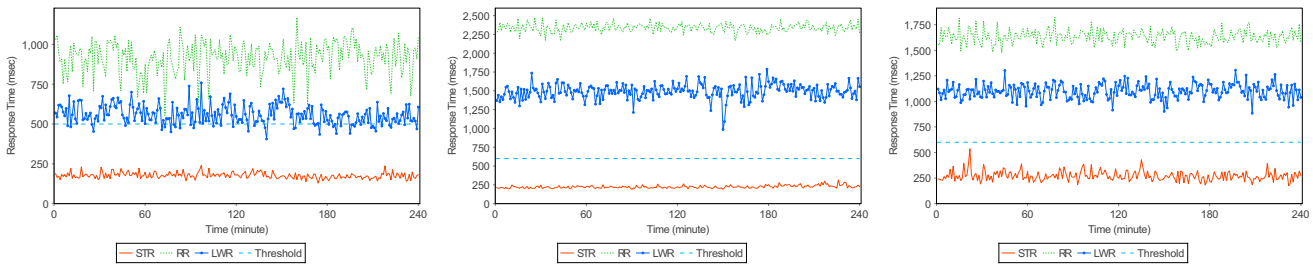


Fig. 6: Total throughput of the cluster

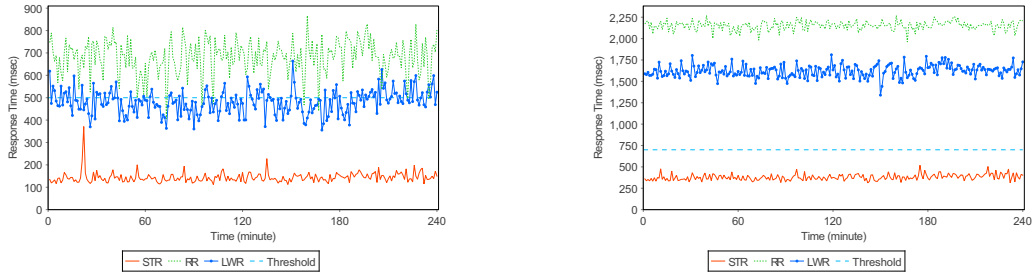
Figure 6 shows the total throughput of the cluster, which is the sum of the actual throughput on each servers. It can be inferred that cluster with STR strategy achieves better server throughput than other approaches under the same workload, thus has a higher resource utilization. However, the generated workload is heavy for LWR and RR policies and extra servers are needed to provide the same performance as STR.

VI. Related Work

Numerous researches have been studied about load balancing algorithms on a web server cluster, which can be classified as content-blind and content-aware algorithms [17]. A content-blind load balancer works at the low layer of the OSI model and is unaware of the application information contained in the requests. On the other hand, a content-aware load balancer works at the application layer and parses the content of requests, such as request URL and cookies. In addition, load balancing algorithms can be classified into static and dynamic algorithms. The static algorithm consistently follows a preset rule to dispatch requests. On the other hand, dynamic algorithms make decisions for dispatching according to the servers' load, hence demand for periodically monitoring and

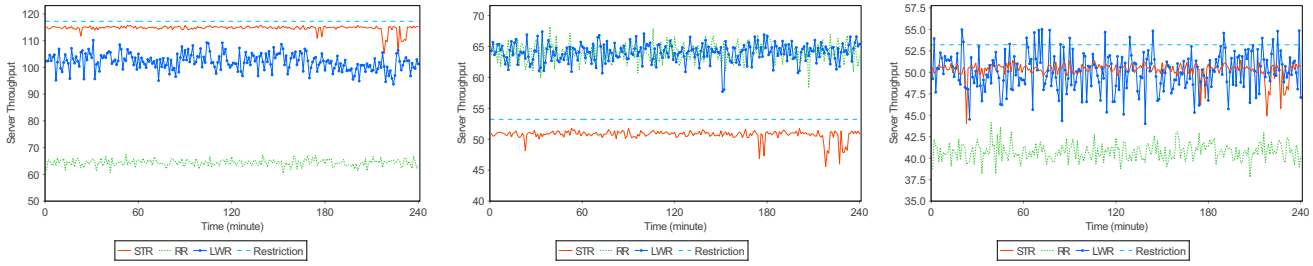


(a) Response time of APP A (b) Response time of APP B (c) Response time of APP C

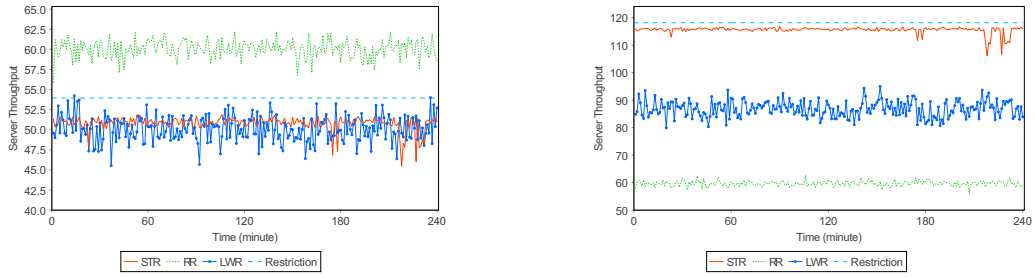


(d) Response time of APP D (e) Response time of APP E

Fig. 7: Response time of APP A-E



(a) Throughput on Server A (b) Throughput on Server B (c) Throughput on Server C



(d) Throughput on Server D (e) Throughput on Server E

Fig. 8: Throughput on Server A-E

gathering load information on each server. Dynamic algorithms usually have higher performance but are more complex in contrast with static ones.

In recent years, QoS-aware load balancing algorithms, which belong to content-aware algorithms, have drawn a lot of attention, most of them trying to maximize throughput or minimize the mean response time of certain types of requests. Authors in [7] propose a QoS-aware algorithm named APRA, the basic idea is that different type of requests have different processing priorities according to their QoS demands, thus requests with higher priority are more likely to be served. Each type of requests has a corresponding waiting queue that implements the RR policy, and has a throughput restriction on each server according to the prediction of workloads.

In [8] the authors propose the ORBITA approach to guarantee that the response time would not be higher than an

acceptable threshold, which dispatches requests to separate servers according to their execution time. Therefore, requests with short execution time are always been served, which have a better performance on throughput and the mean response time when the execution time distribution of requests is heavy-tailed.

Although the QoS-aware load balancing algorithms mentioned above works well in single-application environment, they are not suitable in multi-tenancy environment because they do not concern about the mutual intervention among applications on the same server. Therefore, authors in [18] [19] suggest that the QoS of requests is guaranteed by performance isolation between tenants in order to eliminate their competing for shared resources. The scheduling policy is that tenant with lower resource usage gets higher priority for request servicing, which is a dynamic strategy and guarantees the QoS of tenants specified in SLAs by first guarantees their usage of resources. However, this strategy requires tenant-aware monitoring for resource usage information, which would be inaccurate sometimes and needs extra system resource.

In order to reduce the communication between the load balancer and the back-end servers, Yi Lu et.al [20] propose a novel dispatching method for scalable cloud named JIO, which includes a set of dispatchers with a corresponding queue of idle servers. Each server will notify a randomly chosen dispatcher as soon as it becomes idle. The incoming job will be dispatched to the server in the idle queue and the server will be removed afterwards.

VII. Conclusion

In PaaS multi-tenancy environment, applications are deployed on the same server to improve resource utilization. Most of the QoS-aware load balancing policies are not suitable in multi-tenancy environment since they are not aware about the mutual intervention among applications within the server. Thus, some applications' response time would exceed their threshold due to the higher throughput on the server. In this work, we propose a new load balancing policy named STR, which dispatches requests according to the throughput restriction based on the M/G/s/s+r queueing model. On the basis of our Service4All platform, it was experimentally proved that STR can guarantee the response time of each application and achieves even better server throughput in contrast with RR and LWR, thus improves resource utilization.

As future work we intend to improve the throughput restrictions as we choose the minimum traffic intensity that can be further optimized by workload prediction: we will choose a higher traffic intensity when the load of application with minimum traffic intensity is empty in the next time slot. We also intend to add more competitive factors that applications on the same server are not only CPU-intensive, but also memory-intensive or database-intensive. This includes how to model the problem properly and how to construct the throughput restriction equation of each server.

Acknowledgment

This work was supported by National Natural Science Foundation of China (No. 61370057, No. 61103031), China 863 program (No. 2012AA011203), A Foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. 201159), Beijing Nova Program (No. 2011022) and Specialized Research Fund for the Doctoral Program of Higher Education (No. 20111102120016).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," EECS Department, U.C. Berkeley, Tech. Rep., February 2009.

[2] G. Pallis, "Cloud computing: The new frontier of internet computing," *IEEE J. of Internet Computing*, vol. 14(5), pp. 70–73, 2010.

[3] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle, "Multi-tenant soa middleware for cloud computing," in *Proceedings of the 3rd IEEE International Conference on Cloud Computing*, Miami, FL, USA, July 2010.

[4] Heroku, "https://www.heroku.com."

[5] CloudBees, "http://www.cloudbees.com."

[6] S. Sharifian, S. A. Motamedi, and M. K. Akbari, "A content-based load balancing algorithm with admission control for cluster web servers," *Future Generation Computer Systems*, vol. 24(8), pp. 775–787, 2008.

[7] K. Gilly, S. Alcaraz, C. Juiz, and R. Puigjaner, "Service differentiation and qos in a scalable content-aware load balancing algorithm," in *Proceedings of the 40th Annual Simulation Symposium*, Norfolk, Virginia, USA, March 2007.

[8] L. F. Orleans and P. N. Furtado, "Fair load-balancing on parallel systems for qos," in *Proceedings of the 2007 International Conference on Parallel Processing*, Xi-An, China, September 2007.

[9] C. Oliveira, "Load balancing on virtualized web servers," in *2012 41st International Conference on Parallel Processing Workshops*, September, 2012, pp. Pittsburgh, PA, USA.

[10] Service4All, "http://www.service4all.com.cn."

[11] V. Sundarapandian, *Probability Statistics And Queuing Theory*. PHI Learning, 2009.

[12] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain," *The Annals of Mathematical Statistics*, vol. 24(3), pp. 338–354, 1953.

[13] M. Harchol-Balter, M. Crovella, and C. Murta, "On choosing a task assignment policy for a distributed server system," *Journal of Parallel and Distributed Computing*, vol. 59(2), pp. 204–228, 1999.

[14] J. M. Smith, "M/g/c/k performance models in manufacturing and service systems," *Asia-Pacific Journal of Operational Research*, vol. 25(4), pp. 531–561, 2008.

[15] T. Kimura, "A transform-free approximation for the finite capacity m/g/s queue," *Operations Research*, vol. 44(6), pp. 984–988, 1996.

[16] A. Lee and P. Longton, "Queueing process associated with airline passenger check-in," *Operations Research Quarterly*, vol. 10, pp. 56–71, 1959.

[17] K. Gilly, C. Juiz, and R. Puigjaner, "An up-to-date survey in web load balancing," *World Wide Web*, vol. 14(2), pp. 105–131, 2011.

[18] X. H. Li, T. C. Liu, Y. Li, and Y. Chen, "Spin: Service performance isolation infrastructure in multi-tenancy environment," in *Proceedings of the 6th International Conference on Service-Oriented Computing*, Sydney, Australia, December 2008.

[19] S. Walraven, T. Monheim, E. Truyen, and W. Joosen, "Towards performance isolation in multi-tenant saas applications," in *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, Montreal, Quebec, Canada, December 2012.

[20] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. Greenberg, "Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services," *Performance Evaluation*, vol. 68(11), pp. 1056–1071, 2011.