Service Differentiation in Multitier Data Centers

Kostas Katsalis
University of Thessaly, Greece
Email: kkatsalis@uth.gr

Georgios S. Paschos
LIDS Lab
MIT MA, USA
Email: gpasxos@mit.edu

Leandros Tassiulas
CERTH-ITI
University of Thessaly, Greece
Email: leandros@uth.gr

Yiannis Viniotis
Department of ECE
NCSU, USA
Email: candice@ncsu.edu

***Abstract*—In this paper, we study the problem of resource allocation in the setting of multitier data centers. Our main motivation and objective is to provide applications hosted in the data center with different service levels. In such centers, there are several mechanisms the designer can use to achieve such objectives. We restrict our attention to CPU time at the service tier as the resource; the objective we consider is service differentiation, expressed as allocating prespecified percentages of this resource to applications. Then, mechanisms at the designer's disposal to provide desired service differentiation include the triplet of load balancing through the switch fabric, enqueueing at a server and scheduling at a server. We focus on the enqueueing component of control mechanisms. We provide, through analysis and simulations "rules of thumb" for situations where simple enqueueing policies can provide service differentiation.**

Keywords*—*Data Centers, Service Differentiation, Resource Provisioning

I. INTRODUCTION

Although the well known datacenter multitier architecture still remains the basis of the datacenter network design, Software Defined Networking (SDN) enforces a new approach in the way we build datacenter networks or use virtualization technologies. A main argument in SDNs is that every single hardware or software component must be a service or a service handler/receiver/producer and it was only recently that large vendors decided to use powerful middleware [1] in order to offer services optimization. In large scale deployments for applications like banking transactions, specialized middleware is now widely used as a service accelerator and optimizer. For example the VECSS platform [2] was developed jointly by VISA and IBM and is used to process over than 8,000 banking transactions per second, with over than 500 different transaction types. Specialized middleware can act as the connecting link between SDNs and Application Delivery Networks, having a central role in future datacenter designs.

In Figure 1, a typical multitier datacenter architecture is presented. The key elements of this design, depicted as a three-tier core-distribution-access architecture, are the switch fabric, the preprocessing tier and the final processing tier [3] (depending on the application, not all requests require preprocessing). Core router/switches accept external traffic through the Internet or a VPN and they route this traffic through the fabric to one of several servers in the preprocessing tier. Servers in the preprocessing tier are typically specialized middleware devices that prepare requests for processing in the servers housed at the final processing tier. Typical functions at the preprocessing tier may include XML/XSLT transformations, logging and security/authentication services. After preprocessing, requests are forwarded for service at a server in the final processing tier. This is the place where the "main business logic" is executed.

The datacenter hosts several applications/services and the traffic entering the datacenter is naturally classified into classes - called *Service Domains* (SD). How CPU time is allocated to these domains clearly affects the performance seen by the application requests. In typical Service Level Agreements (SLAs), metrics regarding this performance are included and expressed in various ways. In this paper, we focus on CPU utilization as one of these metrics. For simplicity and clarity of presentation, we consider CPU utilization of the servers in the preprocessing tier only. As an example, an SLA with CPU utilization metrics can be described as follows: guarantee 20% of total (preprocessing tier) CPU power to domain A and 80% to domain B.

Three elements have an effect on how CPU time is allocated to requests of a domain and thus can determine whether the SLA can be honored or not. An administrator can control them in order to provide service differentiation to the various domains. In Figure 1, we label the main available control actions that an administrator can use: First, an arriving request must be routed through the fabric to one of the servers; we call this the *load balancing* control. Once it is sent to a server, a request must be enqueued in a buffer; we call the design of the buffering scheme the *enqueueing* policy. This is the second control at the disposal of the administrator. The third control is the CPU *scheduling* inside the servers, i.e., deciding from which queue to forward a request for processing.

Note that the controls are distributed: two controls are implemented in the server (but we have many of them in the tier) and one is implemented in the switch fabric (at one or more tiers therein). Clearly *all* three controls have an effect on what percentage of the CPU time in the cluster of servers a given domain can have. Design/analysis of a distributed, end-to-end control policy is out of the scope of this study. In this paper, we consider fixed load balancing and scheduling policies and focus on the effect of the enqueueing part of the triplet. Our contribution is twofold (a) we propose a simple design for the enqueueing policy that is geared towards distributed implementation: it has low communication and processing overhead, and, (b) we provide "rules of thumb" for when the design can achieve the SLA. We base these rules on both analysis and simulation.

In Section II, we present the motivation for this study and related work. In Section III, we define the system model and state the SLA and the problem formally. In Section IV, we present the proposed policies; we evaluate them in Section V and provide rules of thumb in Section VI. We conclude the paper in Section VII.

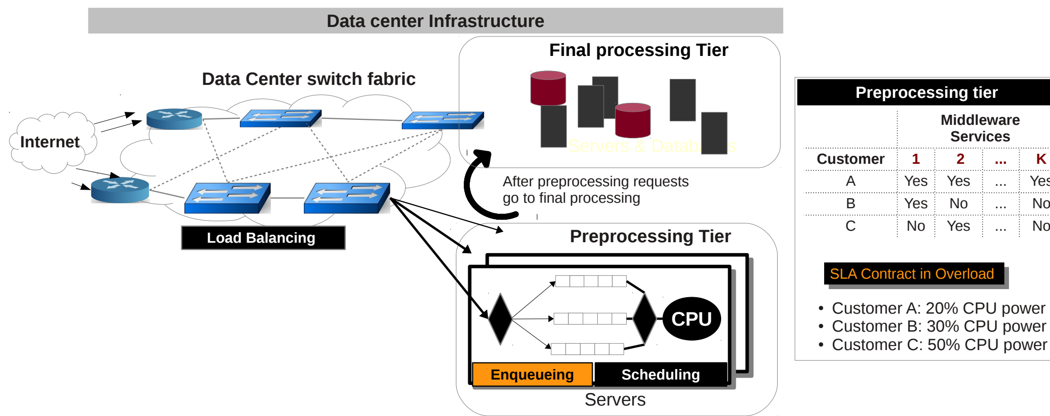


Figure 1. Multitier Architecture

II. Motivation and Related work

In enterprise (data center) SLAs, performance objectives are described in terms of several and varied metrics, such as cost, response time, availability of resources, uptime, and losses. Meeting such SLAs requires careful management of data center resources and is a fine balancing act of policies for short, medium and long-term resource allocation.

When an "overload" occurs, typical SLAs define a different, usually simplified set of performance objectives. For example, under overload conditions, it is very difficult to provide response time or loss guarantees. A common example of an SLA in overload conditions is one that focuses on simply stated CPU utilization metrics; we describe a specific one in the next section. Such simplified SLAs have the additional advantage of not requiring knowledge about the arrival and service statistics of the domain.

In the field of load balancing policies, Bernoulli splitting is a simple mechanism that has been studied under multiple variations (e.g., [4]). Enqueueing mechanisms have been extensively used in the past for providing QoS guarantees [5], while Multiclass-multiqueue systems studies have been studied for token based systems [6].

Scheduling policies have been mainly used to control response time metrics; work in this space typically assumes arrival or service knowledge [7] or makes use of complex feedback mechanisms [8]. In prior work [9], we proved that static scheduling policies fail to provide arbitrary percentages of CPU utilization for every domain. Instead, dynamic Weighted Round Robin scheduling can be efficiently used in overload conditions to provide the desired differentiation given that an ample number of queues is available - typically one queue per service domain [9], [10].

III. Problem Statement

A. System model and assumptions

The system model is presented in Figure 2. We begin by omitting the server tier and collapsing the datacenter network fabric into a load balancer function. Modeling the routing through the switch fabric as a single load balancer is sufficient, since we focus on the enqueueing policies in this study; which (core, aggregation or access) switch(es) were responsible for forwarding the traffic to a preprocessing tier server is irrelevant.

The load balancer is responsible for distributing incoming traffic from a set $D = \{1, ..., d\}$ of service domains into a cluster of $N = \{1, ..., n\}$ servers, each with a CPU of capacity μ. Every middleware server is modeled as Multiclass-multiqueue system (MCMQS), with $\mathcal{M} = \{1, ..., m\}$ the set of queues that is same for all the servers. We assume that $m < d$, i.e., there are not enough queues available to separate traffic belonging to different service domains. This is a reasonable assumption in data centers that offer SaaS, IaaS or PaaS services.

Finally, we assume that the incoming traffic for domain i follows a general arrival and service process with (unknown) arrival rates λ_i and service rates $1/ES_i$, where ES_i is the mean service time of domain i. Also we assume non preemptive service for the CPU operation. We assume that signaling and feedback across the servers take negligible time, however we note that the design of our control system is tailored to minimize these effects.

B. Control policy definition

A control policy π is a rule according to which control actions are taken at time t. We identify different control policies regarding different control points inside the network.

1) *Load balancing policy* π_r that defines a *forwarding action* $r(t_r)$. Say t_r is the time instant that the load balancer must forward an incoming request to the cluster of dedicated servers. The action space is the set $N = \{1, ..., n\}$ of all the servers and $r(t_r) = i \in N$ at time t_r if server i is selected.
2) *Enqueueing policy* π_q that defines *Enqueueing Action* $q(t_q)$. Say t_q is the time instant that a server accepts a request from the load balancer. An enqueueing action determines the queue to which the request is forwarded. The action space is the set $M = \{1, ..., m\}$ of all the available queues and $q(t_q) = i \in M$ at time t_q if queue i is selected.
3) *Scheduling policy* π_s that defines *Scheduling Action* $a(t_s)$. Say t_s is the time instant that a server CPU finishes request execution and is ready to accept a new request for service. The action space is the set

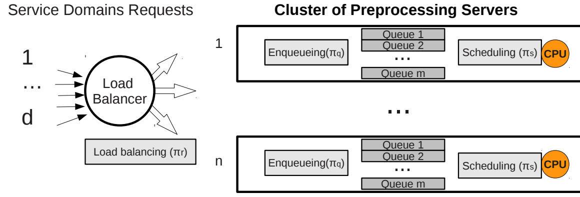


Figure 2. System Model

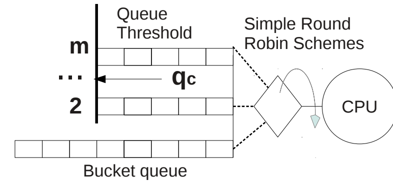


Figure 3. Queue Structure

$M = \{1, ..., m\}$ of all the available queues and so $a(t_s) = i \in M$ at time t_s if queue i is selected.

The vector (π_r, π_q, π_s) is collectively referred to as the control policy π.

For server j, we define the function $f_i^j(t, \pi)$ as the amount of time in $[0, t)$ that the server CPU was allocated to domain i, under policy π. Then for the total CPU power of the cluster of servers, we define the **total utilization** for domain i under policy π as:

$$u_i(\pi) \triangleq \liminf_{t\to\infty} \frac{\sum_{j=1}^{n} f_i^j(t, \pi)}{n \cdot t} \quad (1)$$

In the above definition we use liminf since we don't know a priori that the policy will reach steady state.

C. Problem Statement

The formal definition of the objective we study is the following: *Let $0 \le p_i \le 1$ be CPU utilization percentages defined in the SLA for every domain i. Design a policy π that will achieve the following objective:*

$$u_i(\pi) = \min\{\lambda_i \cdot ES_i, p_i\}, \forall i \in D \quad (2)$$

Roughly speaking, the SLA in Equation 2 states that each domain must be given a predetermined CPU time percentage over a large time period, unless the request rate is not sufficient for this, in which case it is enough to serve all requested traffic from that domain.

IV. Proposed Policies

A thorough investigation of what policy π achieves the SLA in Equation 2 is out of the scope of this paper. The reason is twofold. First, because of the joint control definition, the three control actions depend greatly on each other. For example the performance space of a dynamic Weighted Round Robin scheme used as the scheduling policy π_s is in correlation with the queueing dynamics that are dictated by the enqueueing policy π_q. The second reason is there is a large set of system configurable parameters like the number of queues, or the number of service domains that greatly affect the performance space.

For these reasons, in this paper we focus only on the enqueueing policy π_q. We investigate the single server case ($n = 1$) and we gain the insights into the queueing dynamics effects in the system performance. The accompanying scheduling policy will be plain Round Robin. The aim is to avoid feedback signals analysis between different tiers in the data center and feedback signals for centralized control of the cluster of servers. In addition, with simple scheduling we avoid designing scheduling algorithms that require knowledge of traffic statistics and we avoid the correlation analysis between enqueuing and scheduling decisions.

A. Proposed enqueueing policy: BQS (Bucket Queue System)

Queueing configuration: The server is configured to accept traffic from all the domains in a set of $\mathcal{M}$ queues. Set for all queues $i : 2 < i < m$ a queue limit q_c. We will use this threshold to limit the corresponding queue sizes, while queue $i = 1$ is allowed to grow without constraints. We call this queue the *bucket queue* (*B* queue); the configuration can be seen in Figure 3.

Operation: In the case where an incoming request from domain i must be enqueued because the CPU is busy upon its arrival, the following actions take place: If the domain is underutilized and the minimum queue size does not exceed the limit q_c, the policy forwards the request to the queue with the minimum queue size. In all other cases the request is forwarded to the bucket queue. The algorithm operation is formally described as Algorithm 1.

Focus on the server CPU operation and let t_k denote the time instant that the CPU finished servicing a request and that the next request it will receive service comes from domain i. We define $u_i(t_k, BQS)$ to be the CPU utilization domain i received up to time t_k under policy BQS. Between two successive service events t_k and t_{k+1} the following is true: $t_{k+1} = L_k + Z_i^k + t_k$, where $L_k \in R$ is a rv denoting the CPU idle time between the service events and Z_i^k is a rv of the service time the request of domain i received within this interval, while $Z_{j\neq i}^k = 0$. The CPU utilization performance now can be expressed with the following closed recurrent form: $u_i(t_{k+1}, BQS) = \frac{t_k}{t_{k+1}} u_i(t_k, BQS) + \frac{Z_i^k}{t_{k+1}}, \forall i \in D$. Clearly BQS policy and the accompanying scheduling policy that will be adopted (Round Robin in our case), at any instant t_k affect the Z_i^k rv and so the utilization every domain will receive.

Intuition: For overload conditions, when $\sum_{i=1}^{d} \lambda_i/\mu \ge 1$, the *B* queue is unstable while the remaining $m - 1$ queues remain stable. This way, traffic injected to the stable queues receives guaranteed throughput while traffic injected to the *B* queue only receives a proportion of throughput in relation to what is injected. The policy is using a simple control mechanism to improve the utilization of domains that are left behind their goals while penalizing those that have received more service than agreed.

V. Policy Evaluation

With respect to simplicity of implementation, we examine the triplet $\{RR, BQS, RR\}$ i.e., load balancing and CPU scheduling are round robin and enqueueing is BQS. Although, since we have 1 server, load balancing has no meaning.

Algorithm 1 BQS Algorithm

t_k : *enqueueing decision instance*
$Q_j(t_k)$: *the queue size of queue j in time.*
Calculate $u_i(t_k, BQS) < p_i, Q_j(t_k)$
if $u_i(t_k, BQS) < p_i, Q_j(t_k) \leq q_c$ **then**
  $q(t_k) = \arg\min_j Q_j(t_k)$
else
  $q(t_k) = 1$
end if

A. Theoretical considerations

1) Setup: We will consider a single server with m queues served by a CPU of capacity μ in a round robin fashion, such that each queue receives service with rate μ/m. The $m-1$ queues are bounded (a job is routed to them only if their backlog is below the threshold q_c) and one is unbounded. We will study the case of two service domains in order to derive the conditions for SLA achievability, i.e. the set of (p_1, p_2) targets for which the SLA is achievable under given -but unknown- conditions $(\mu, \lambda_1, \lambda_2)$. Generalizations to many domains and many servers are left for future work.

Let u_1, u_2 denote the utilizations and T_1, T_2 the throughputs, which are related in the following way $u_i = \frac{T_i}{\mu}$. Also, let $a_1, a_2 \in [0, 1]$ be the long-term average traffic splits of the arrivals, i.e. $a_1\lambda_1$ is the traffic directed to the *B* queue and $(1 - a_1)\lambda_1$ the traffic directed to the $m - 1$ queues for the service domain 1. In what follows, we will attempt a fluid analysis omitting the details regarding the arrival processes and avoiding the complications of a discrete time analysis.

2) Analysis: We derive necessary and sufficient conditions for the feasibility of the SLA target, under the condition that the queues are configured as explained above. However, we do not show that the algorithm indeed converges to the proper routing coefficients. The fact that the algorithm can achieve this feasibility region will be shown by simulations. We have two cases that regard the comparison $\lambda_1 + \lambda_2 \lesseqgtr \mu$.

When the system is stable (i.e. $\lambda_1+\lambda_2 \leq \mu$), the SLAs are always achieved since the throughput of each user equals what is injected into the system, thus the first term of the minimum function in eq. (2) is always achieved.

When the system is unstable, (i.e. $\lambda_1 + \lambda_2 > \mu$), the *B* queue is unstable but the $m - 1$ queues remain stable. We will inspect two further cases, a) when both domains request more traffic than their corresponding target $\lambda_i > \mu p_i$ and b) when one of the two requests more, but the other less. We also omit c) the symmetric to b and d) the case where none requires more than the target, which contradicts the instability condition.

Case a: Assume $\lambda_1 > \mu p_1$ and $\lambda_2 > \mu p_2$. Note that the flow is conserved in the $m - 1$ queues, which gives

$$(1 - a_1)\lambda_1 + (1 - a_2)\lambda_2 = \frac{m-1}{m}\mu. \quad (3)$$

The service in *B* queue is proportionally allocated to the two domains:

$$\mu_1 = \frac{a_1\lambda_1}{a_1\lambda_1 + a_2\lambda_2}, \quad \mu_2 = \frac{a_2\lambda_2}{a_1\lambda_1 + a_2\lambda_2}$$

and the utilization of domain i should be

$$\begin{aligned} u_i &= \frac{(1-a_i)\lambda_i + \frac{a_i\lambda_i}{a_1\lambda_1+a_2\lambda_2}\frac{\mu}{m}}{\mu} \\ &\overset{(3)}{=} \frac{(1-a_i)\lambda_i(\lambda_1+\lambda_2-\mu) + \lambda_i\frac{\mu}{m}}{\mu(\lambda_1+\lambda_2-\frac{m-1}{m}\mu)}, \quad i = 1, 2. \quad (4) \end{aligned}$$

The following are a set of necessary and sufficient conditions for the SLA to be satisfied under the stated conditions (instability and both users providing sufficient arrivals):

$$\begin{cases} p_1 = u_1 \\ p_2 = u_2 \\ (3) \end{cases}$$

Dividing the first two, and using (4) and $p_2 = 1 - p_1$

$$\frac{p_2}{p_1} = \frac{\mu(\lambda_1+\lambda_2-\frac{m-1}{m}\mu)}{(1-a_1)\lambda_1(\lambda_1+\lambda_2-\mu)+\lambda_1\frac{\mu}{m}} - 1, \quad a_1 \in [0,1].$$

Denote $M \doteq \max_{a_1\in[0,1]} \frac{p_2}{p_1}$. Clearly this is achieved by $a_1 = 1$, in which case

$$M = \frac{\mu + m(\lambda_1 + \lambda_2 - \mu)}{\lambda_1}. \quad (5)$$

Note that M is the maximum achievable ratio of target utilizations under the conditions above, and it can be achieved by an omniscient randomized enqueueing policy which selects properly a_1, a_2.

Case b: W.l.o.g. assume $\lambda_1 < \mu p_1$ and $\lambda_2 > \mu p_2$, thus an SLA-satisfying enqueueing solution will yield $(u_1, u_2) = (\lambda_1/\mu, p_2)$. Note, that if non-negligible fluid from domain 1 is routed to the bucket, then the throughput of domain 1 will be less than what sent, in which case the SLA has failed. Thus, $a_1 = 0$ and we obtain a first condition:

$$\lambda_1 \leq \frac{m-1}{m}\mu.$$

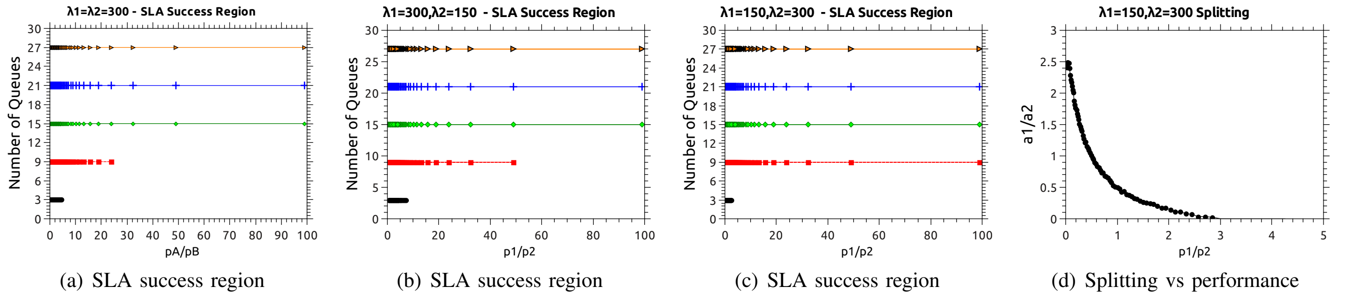
The domain 2 will receive the remaining CPU allocation, thus the SLA is satisfied if $p_2 \leq \frac{\mu-\lambda_1}{\mu}$, from which we conclude

$$\frac{p_2}{p_1} \leq \frac{1-\frac{\lambda_1}{\mu}}{\frac{\lambda_1}{\mu}} = \frac{\mu-\lambda_1}{\lambda_1} = M \quad (6)$$

which does not depend on m.

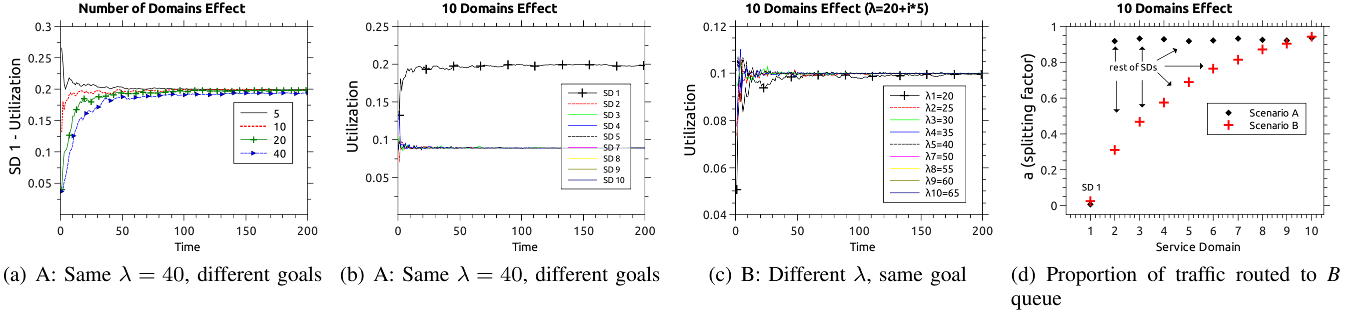
B. Verification of the theoretical considerations

In Figure 4 we present the maximum ratio of p_1/p_2 for which the SLA is satisfied, in the following scenario: a single server receives traffic from two domains. The service rate of the server CPU is set equal to $\mu = 200$ requests per time unit (we assume that time is a dimensionless quantity). The criterion we apply in order to note SLA success is $u_i(\pi) \geq 0.95 \min\{\lambda_i/\mu, p_i\}, \forall i \in D$ which is the objective defined in equation 2 plus a $\pm 0.05\%$ tolerance interval. In Figures 4(a), 4(b) and 4(c) we present how the SLA success region is affected by increasing the number of queues. The main outcome is that when m, the number of queues, is sufficiently large the algorithm is able to achieve the SLA even for extreme CPU differentiation goals (e.g., 90-10%). The



(a) SLA success region (b) SLA success region (c) SLA success region (d) Splitting vs performance

Figure 4. BQS performance for domain requesting 0.7-0.3 of CPU Utilization - $\mu = 200$ requests



(a) A: Same $\lambda = 40$, different goals (b) A: Same $\lambda = 40$, different goals (c) B: Different λ, same goal (d) Proportion of traffic routed to B queue

Figure 5. Number of Domains effect ($\mu = 200$ requests)

value of m that guarantees extreme targets depends also on the relation between λ_i and μ, when the system is unstable. When the system is stable, the targets are always achieved.

For example, as we can see in Figure 4(a), with 3 queues the SLA is achieved for a ratio $1/6$, meaning that an SLA $p_i = \{0.86, 0.14\}$ can be satisfied, while with 9 queues this ratio is $1/25$ meaning $p_i = \{0.96, 0.04\}$ can be satisfied.

Figure 4(d) clearly validates the theoretical considerations of the previous section. We present the ratio a_1/a_2 for the two domains in the case where $\lambda_1 = 150$ and $\lambda_2 = 300$ requests per time unit (scenario of Figure 4(c)). As we can see, domain 1 takes the maximum CPU share it can receive and the maximum ratio is achieved, when domain 2 forwards all its traffic in the B queue. All the above observations are summarized as a Rule of Thumb 1 in Section VI.

C. Increasing the number of domains

A key advantage of BQS policy is that for a large number of domains, a small number of queues is sufficient in order to meet the objective of equation 2. We present simulation evaluation for two scenarios that a service administrator faces when the system is in overload conditions:

Scenario A: *An increasing number of domains request server resources. Service domain 1 is the most prominent client and the administrator wants to guarantee that during overload periods it receives 20% CPU share while the rest divide the remaining 80% equally.*

Configuration: $\lambda_i = 40$ and $\mu = 200$ for every domain i, while $m = 5$ is the number of queues (including the B queue). In Figure 5(a) we increase the number of domains and in Figure 5(b) we present in more detail the case where $d = 10$. In both figures we can see that BQS clearly meets the objective. Since we are in overload, regardless of the relationship between arrival and service rates and also regardless of the range of desired percentiles, the enqueueing algorithm satisfies the SLAs, i.e., domain 1 always receives the requested 20% while the remaining domains take their equal share. In Figure 5(d) we can see that in scenario A (black bullets), a_1 factor of domain 1 is approximately equal to zero, while for the rest of domains it is almost equal to 0.9. This means that B queue serves about 10% of domain 1 traffic and the majority of other domains traffic.

Scenario B: *Again an increasing number of domains request server resources. The administrator wants to guarantee fair allocation of CPU power to all the domains, despite any arrival rate diversifications.*

In Figure 5(c) BQS policy also meets the objective and fair allocation is provided in the scenario where the arrival rate of every domain is different. The simulation configuration is $\lambda_i = 20 + i * 5$ and $\mu = 200$ domain while again there are $m = 5$ queues. As we can see in figure 5(d) for the B scenario case (red crosses), the mechanism to achieve the SLA is again the operation of the B queue. If the traffic from a domain is higher than the others, a higher percentage of its requests is served by the B queue to keep allocating CPU cycles fairly.

Concluding remark: Besides the above paradigms, an extended set of simulations was performed, for various scenarios with increasing number of domains, number of servers and arrival and service rate variations. The main outcome of the simulation procedure is that a small number of queues is sufficient for the simple proposed control mechanism to be effective in the sense that the majority of SLA configurations can be achieved. In the above theoretical and practical considerations analysis, we presented the performance of the algorithm for the case where the number of servers is $n = 1$. Thorough theoretical investigation for the case where the BQS is applied

in a cluster of servers will be presented in future work.

VI. Practical Considerations

Due to the large number of parameters that affect the algorithm performance and the large number of trade offs existing, we summarize the evaluation procedure in the following rules of thumb that can be used by a system administrator. Say d is the number of domains, λ_i is the expected arrival rate of domain i and μ is the service rate of the CPU:

Rule 1: *What is the maximum ratio we can achieve and how many queues are needed?* The answer comes directly from equations 5 and 6. An administrator can make a priori an arrival and service rate estimation of the expected traffic and apply these equations. These equations will show if the desired utilizations defined in the SLA contract will be satisfied. If the estimations are wrong, real time corrections with adding/removing of queues can fix the ratio M to the desired value. In the case where $d > 2$, the generalization of equation 5 yields that the utilization domain i will receive under overload is:

$$u_i = \frac{(1-a_i)\lambda_i + \frac{a_i\lambda_i}{\sum \lambda_i - \frac{m-1}{m\mu}}\mu}{m\mu} \quad (7)$$

This is an upper bound that an administrator can use to jointly design the SLA for all the domains. This way our formula depends only on a_i which is better, but not conclusive. The administrator can use it as a parameter to bound u_i by setting $a_i = 1$ or $a_i = 0$.

Rule 2: *What queue limit to set?* In saturated conditions if we don't use the queue limit q_c, all queues would be unstable and thus no speeding up could be applied for the underutilized customers. The queue limit q_c can be set to any "small" value an administrator chooses to avoid idle time.

With the above rules a system administrator is able to provide predefined CPU percentages to every customer in cases of overload without any scheduling or load balancing concerns and without knowing a priori (or posteriori) any arrival or service statistics. The main advantages of our approach are that no requests are dropped and that a limited number of queues is sufficient to provide utilization defined in enterprise SLAs. The only necessary tool applied is CPU monitoring. Furthermore, in actual systems with injective allocation of domain/queue, BQS algorithm could be easily implemented if *B* queue was added as a new software component without disrupting the overall queue structure.

VII. Conclusions & Future work

In this paper, we studied the problem of resource allocation in the setting of multitier data centers. The objective was to design enqueueing policies to provide desired service differentiation in overload situations. The SLA was defined in terms of CPU time. Through analysis and simulations we provided "rules of thumb" for situations where BQS, a simple enqueueing policy can provide service differentiation. The BQS policy provides the ability to "increase/decrease the priority" of traffic that received less/more service than the goal in the SLA by diverting traffic to limited or unlimited size queues. This diversion is done dynamically, based on the current CPU utilizations. In future work, we plan to perform further theoretical investigation of this policy and evaluate its speed of convergence.

Acknowledgements

This work is financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) Research Funding Program: Heracleitus II - Investing in knowledge society through the European Social Fund.

References

[1] IBM Redbooks, "DataPower Architectural Design Patterns: Integrating and Securing Services Across Domains", 2008

[2] IBM and VISA "Vecss platform", Smarter planet, April 2011, http://www.ibm.com/smarterplanet/banking

[3] Arregoces M. et al, "Data Center Fundamentals", Cisco Press, 2004

[4] Hyyti E.; Virtamo J.; Aalto S.;Penttinen A., "M/M/1-PS queue and size-aware task assignment", Performance Evaluation, v.68, Issue 11, 2011

[5] Andelman N.; Mansour Y.;Zhu A.;, "Competitive queueing policies for QoS switches", In Proceedings of ACM-SIAM symposium on Discrete algorithms (SODA '03). Philadelphia, PA, USA, 761-770.

[6] Takagi H., "Queuing Analysis of Polling Models", IBM Research, Tokyo Research Laboratory, Japan, 1988

[7] Addad B. et al, "Analytic Calculus of Response Time in Networked Automation Systems", IEEE Trans. on Automation Science and Engineering, vol.7, n.4, Oct. 2010

[8] Sha L.; Liu X.; Lu Y.; Abdelzaher T. , "Queueing model based network server performance control", IEEE RTSS Symposium, 2002

[9] Katsalis K.; Paschos G.; Tassiulas L.; Viniotis Y. , "Dynamic CPU Scheduling for QoS Provisioning", IFIP/IEEE Information Management (IM), 2013

[10] Lieshout P.; Mandjes M.; Borst S.; , "GPS Scheduling : Selection of Optimal Weights and Comparison with Strict Priorities", SIGMetrics/Performance06, Saint Malo, France, 2006.