A SERVICE DIFFERENTIATION ALGORITHM
For Clusters of Middleware Appliances

Mursalin Habib, Yannis Viniotis
Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695, U.S.A.

Bob Callaway, Adolfo Rodriguez
IBM, Research Triangle Park, NC 27709, U.S.A.

Keywords: Middleware, Service Oriented Architecture, Service Differentiation, Closed Loop Feedback.

Abstract: Service oriented architectures (SOA) and XML-based Web Services have become the technology of choice in enterprise networks. These networks support multiple services and are typically architected in multiple computing tiers, with a main service tier for the business logic and a separate, "offload" tier, for, say, the CPU-intensive XML processing. The offload tier is typically populated by clusters of middleware appliances, usually hardware-assisted devices that are optimized for their tasks. Service differentiation refers to the generic problem of managing the enterprise network resources in order to achieve desired performance objectives on a per service basis. In this paper, we define a SAA/SDA (Service Activation Algorithm/Service Deactivation Algorithm) that manages the CPU allocation in the appliance tier, in order to provide service differentiation. The main design objective of SAA/SDA is to overcome the disadvantages of the present known, static solutions. We analyze the performance of SAA/SDA via simulations.

1 INTRODUCTION

Service oriented architectures (SOA) have become the technology of choice for satisfying many business goals in terms of flexibility, software reuse, and addressing complexity (Erl, 2004), (Michael Huhns, 2005). A way of adopting SOA is through exposing functionality as Web Services. These services leverage the ubiquitous nature of XML as a universal message format; however, this choice often imposes increased computational overhead due to XML parsing. For this reason, enterprise network administrators deploy specialized, hardware- assisted appliances for XML processing. These appliances, called middleware appliances or SOA appliances, are positioned on the edge of the enterprise network, as a separate tier "in front of" the service tier. They are generally deployed in multiples to provide sufficient processing power and to meet high availability requirements. Figure 1 depicts an abstraction of such an environment that emphasizes the clustering of appliances and servers into two separate computing tiers.

1.1 Service Differentiation

Typically, an enterprise network supports multiple classes of service requests (also known as *service domains*) (Menascé et al., 2001), (Chandrashekar et al., 2003). For the purposes of this paper, and at a high-level, a service domain corresponds to a deployed application or related applications. *Service differentiation* refers to the generic problem of managing the enterprise network resources in order to achieve desired performance objectives on a per domain basis. For example, resources may include the CPU processing power at the appliance tier and/or the service tier; performance may be defined in terms of throughput for one service domain or average delay for another.

It is up to the enterprise network administrator to properly manage (that is, configure and provision) the system resources together as a collective whole, to achieve service domain differentiation. In this paper, we focus on the issue of managing the "island" of middleware appliances. More specifically, we consider the problem of *Unqualified Service Differentiation* that can be stated as follows: "allocate a desired percentage of the CPU power of the appliances to a given service domain". For example, assuming only three domains SD1, SD2 and SD3, and two appliances with one unit of CPU each, a desired allocation of processing power may be 50%, 30% and 20% respectively.

Habib M., Viniotis Y., Callaway B. and Rodriguez A. (2009).
A SERVICE DIFFERENTIATION ALGORITHM - For Clusters of Middleware Appliances.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 13-20
DOI: 10.5220/0002254300130020

1.2 Mechanisms for Service Differentiation

A variety of mechanisms can be used to effect this allocation. For example, one such mechanism is "priority-based" CPU scheduling of service domains (see for example, (Parekh and Gallager, 1993)). This mechanism requires per domain buffering and is typically used in the server tier. Another one, used more often in inexpensive appliance devices without built-in intelligence (e.g., FIFO buffering for all domains) and without CPU scheduling, is *"activation/deactivation"* of service domains at the gateway: if more CPU resources are needed to achieve its goal, a service domain can be activated at additional appliances; or, more instances of the domain can be activated at the same appliance. Similarly, a service domain can be deactivated from a subset of the appliances if it exceeds its performance goal. In that sense, activation/deactivation of service domains can be seen as an attempt to *alter the rate* at which requests are allowed to enter the system from the gateway. Allocation of CPU resources is controlled *indirectly*, since is is well-known that a service domain with rate λ and average service time ES will achieve a utilization of $\lambda \cdot ES$ (in a stable system).

To the best of our knowledge, there are two known solution approaches for providing differentiated services via activation/deactivation actions, as we describe in section 2.1. In summary, both known approaches result in (a) inefficient use of appliance resources, and, (b) the inability to provide service differentiation. We address both issues in this paper. We describe how we could effect *dynamic* provisioning of service domains amongst a cluster of appliances. This way, unlike the known solutions, service domains are not statically bound to a particular (subset of) appliances.

In summary, the main contribution of our research is an algorithm that, unlike the presently known solutions, has the following advantages: (a) it is capable of providing arbitrary allocation of CPU resources to service domains, thus achieving true service differentiation, (b) it utilizes appliance resources in an efficient manner, and thus it leverages processing whitespace across all appliances, (c) it increases service locality, and, (d) it does not require manual configurations.

The paper is organized as follows. In Section 2, we provide the system architecture and formulation of the problem. In Section 3, we outline the proposed algorithm. In section 4, we summarize the simulation results and answers to the research questions raised.

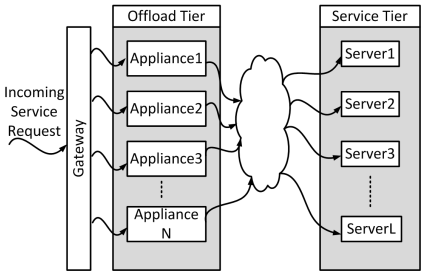

Figure 1: Abstract architecture of a two-tier enterprise system.

2 PROBLEM FORMULATION

The overall architecture of the system under consideration is depicted in Figure 1. Service Requests from clients arrive, via a generic transport network, to be processed in the system. A **Gateway** is the first entry point of the system we are going to consider. The gateway distributes requests to the appliances. **Service Domains** represent the grouping of different service requests. **Servers** process service requests belonging to different service domains. The servers are organized in a "service tier", in the shown architecture. (Middleware) **Appliances** are responsible for pre-processing service requests from different service domains.

The appliances have the capability of buffering requests, in order to accommodate bursty traffic; we assume that they process service requests in a FIFO manner and without preemption[1].

In this paper, we focus on the issue of managing the "island" of middleware appliances. More specifically, we consider the problem of *Unqualified Service Differentiation* that can be stated as follows:

> **Unqualified Service Differentiation:** Provide Service domain m with upto a certain percentage, P_m, of CPU cycles in the appliance cluster.

In typical, commercial SLAs, the percentages may differ, based on whether the system operates under normal or overload conditions. For simplicity, we consider the case of a single condition, hence the name unqualified. We propose an algorithm for achieving Unqualified Service Differentiation in section 3. We present some prior work next.

[1]Even with the presence of a CPU scheduling algorithm inside the appliance, when the number of service domains exceeds the number of buffering classes, service requests within a buffering class are still processed in a FIFO manner.

2.1 Prior Work

As discussed in section 1.2, we consider mechanisms that solve the Unqualified Service Differentiation problem via activation/deactivation actions only. Our motivation for focusing on such mechanisms comes from two reasons: (a) the fact that existing, commercially available appliances utilize this method, and, (b) in systems with large numbers of service domains, multiplexing multiple domains onto the same buffer forces FIFO scheduling. To the best of our knowledge, there are two known solution approaches; both assume the existence of a gateway device (typically an HTTP router/IP sprayer, HRIS) to distribute service load to the appliance cluster. HRIS is a device without deep-content inspection intelligence that simply routes on, say, a URL, and uses IP spraying to send traffic to appliance replicas.

In the first approach, the administrator groups all appliances in a single group and enables them all to process service requests for any given service (Zhu et al., 2001). The fronting IP sprayer would forward a service request to each of the appliances, routing the request to the appropriate service port for service-specific processing. This approach suffers from a number of drawbacks. First, in enabling all service domains on every appliance, it is much more difficult to effect differentiated services across service domains competing for the same appliance resources. While an IP sprayer can effectively spread the load (based on policy) amongst the different appliances, it cannot gauge the effect of a specific service request on CPU and thus cannot provide differentiated service amongst the competing service domains. For example, if the administrator wishes to allocate up to 50% of total CPU to a particular service domain, the system as whole can only hope to evenly spread across the appliances, which, under overload conditions, leads to each service domain receiving 1/3 (33%) of the total CPU. A secondary problem is that it becomes nearly impossible to effect any spatial locality with this solution (Zhu et al., 2001),(Wang et al., 2008).

In the second approach, the administrator may statically allocate a portion of the appliances to each of the service domains. In this case, each appliance is assigned a specific service domain(s) that it will serve. In this way, service requests for a specific service domain are concentrated on specific appliances, thus achieving spatial locality. Further, the administrator can allocate appliances for service domains proportional to the intended capacity (and to some extent priority) for each individual service, thus achieving some level of differentiated service. However, this approach also has a few drawbacks. First, it is difficult to leverage the white space of appliances serving one service for satisfying requests intended for overloaded appliance and its service. That is, under certain conditions, many of the overall system resources may go under-utilized. Second, the allocation process is manual and cannot adapt to changing request rates and prevailing conditions. This could lead to inefficient resource partitioning and ultimately violate intended differentiated service goals (Sharma et al., 2003),(Ranjan et al., 2002),(Zhang et al., 2008).

3 ALGORITHM DESCRIPTION

SAA/SDA is a closed-loop, feedback-based reactive algorithm. It collects periodic performance measurements from the appliances and uses them to alter the rate of the incoming traffic to meet the differentiation goal. To describe the algorithm we need the definitions provided in subsection 3.1.

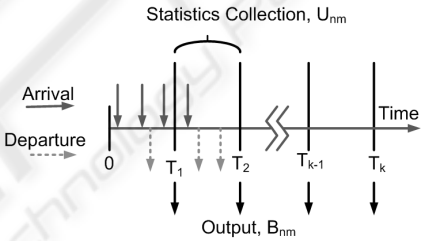

Figure 2: Decision Instances T_k.

3.1 Definitions

The **Provisioning Agent (PA)** is responsible for deciding on activation/deactivation of service domain instances in the appliance cluster. This agent can be implemented as a centralized or distributed application, residing on one or more appliances or a separate compute node. How PA collects the measured statistics from the appliance cluster is out of the scope of this paper.

Decision Instant (T_k) is the k^{th} decision instant at which PA activates/deactivates service domain instances based on the algorithm outcome. As denoted in Fig. 2, at T_k, all the measurements collected in the time interval (T_{k-1}, T_k) are evaluated; activation and deactivation of service domains are enforced. In our simulations, T_k is assumed to form a periodic sequence, for simplicity.

Target CPU % (P_m) is the desired percentage of CPU resources to be allocated to the m^{th} service domain. **Achieved CPU % ($X_m(T_k)$)** is the percentage of the cluster CPU resources obtained by the m^{th} service domain until time T_k.

Down and Up Tolerances DT_m and UT_m: in order to avoid unnecessary oscillations and overhead, when the Achieved CPU % is "close enough" to the Target CPU %, i.e., when

$$P_m - DT_m < X_m(T_k) < P_m + UT_m \quad (1)$$

the service domain is excluded from activation/deactivation.

Utilization Matrix (U_{nm}) is the achieved resource utilization (e.g., total CPU time used) by the m^{th} service domain in the n^{th} appliance, in the time interval (T_{k-1}, T_k).

Instantiation Matrix (B_{nm}) is the number of instances of the m^{th} service domain that should be activated in the n^{th} appliance during the time interval (T_{k-1}, T_k). This is the main decision variable that the PA computes. The mechanism of signalling *HRIS* about the values of B_{nm} and how PA collects the measured statistics from the appliance cluster is out of the scope of this paper.

N is the total **Number of Appliances** in the cluster. M is the **Number of Service Domains** supported by the system.

Groups A and D denote the ranking of service domains. When service domain m is not achieving its Target CPU % (P_m), the PA selects it to be activated in the next decision instant in one or more appliances and thus includes it in Group A. Similarly, when service domain m is allocated more than its Target CPU % (P_m), the PA selects it to be deactivated in the next decision instant in one or more appliances and thus includes it in Group D.

3.2 Algorithm Summary

At each decision instance, at time T_k, $k = 1, 2, \ldots$

1. **Collect measurements** (U_{nm}) from the N appliances.
2. **Calculate the actual percentile of allocated resources** for the M service domains using the iterative equation:

$$X_m(T_k) = \frac{1}{kN}\sum_{n=1}^{N} U_{nm} + \frac{k-1}{k} X_m(T_{k-1})$$

   This equation is a recursive way of calculating the long-term time average of the CPU utilization.
3. **Calculate Thresholding** operations according to Eqn. 1.
4. **Evaluate and Rank Performance** to check if the goal is met. Intuitively, the lower $|X_m(T_k) - P_m|$ is, the "better" the performance of that particular service domain. The service domain is placed in Group A or D as follows. When

$$X_m(T_k) - P_m \geq 0$$

   the service domain meets or exceeds its target and is thus included in Group D. When

$$X_m(T_k) - P_m < 0$$

   the domain misses its target and is thus included in Group A.
5. **Apply Deactivation Algorithm** to deactivate instances of all service domains in Group D as per algorithm SDA (defined in subsection 3.3).
6. **Apply Activation Algorithm** to activate instances of all service domains in Group A as per algorithm SAA (defined in subsection 3.3).
7. **Feedback** these decisions (expressed as values of the matrix B_{nm}) to the gateway.

The intuition and hope is that, during the next interval (T_k, T_{k+1}), the rate of service requests for a domain m will be favorably affected. Activating "more" instances of a service domain will, hopefully, increase the rate at which requests enter the appliance tier. Thus, the domain will see an increase in its share of the cluster CPU resources; note that the increase may not be obtained during the "next" cycle, due to the effects of FIFO scheduling. Similarly, deactivating instances of a service domain will, hopefully, decrease the rate at which requests enter the appliance tier. Thus, the domain will eventually see a decrease in its share of the cluster CPU resources.

3.3 SAA/SDA Activation and Deactivation Algorithm

There is a myriad of choices in how activation and deactivation of service domains can be enforced. We briefly outline only one choice here; due to the lack of space, we omit specifications of what actions should be taken in certain "special cases". For more choices and a more detailed description of implementation ramifications, see (Habib, 2009).

1. (SDA) Deactivate one instance of every service domain in Group D in appliances which run service domains in Group A to free up CPU cycles utilized by domains in Group A.[2]
2. (SAA) Using the instantiation matrix B_{nm}, activate one instance of every service domain in Group A, in appliances which run service domains of Group A.

[2]One of the omitted special cases specifies that deactivation of a domain should not take place if this action leaves the service domain with zero instances active.

Note that both SDA and SAA will result in a change of the values stored in the matrix B_{nm}. As an example, suppose that we have 4 appliances and 5 service domains with target CPU percentages set at $\{35\%, 25\%, 15\%, 10\%, 5\%\}$. Suppose that the initial value for the instantiation matrix is given by

$$B = \begin{bmatrix} 1 & 0 & 10 & 1 & 4 \\ 10 & 3 & 1 & 2 & 4 \\ 0 & 5 & 8 & 4 & 1 \\ 0 & 0 & 2 & 4 & 10 \end{bmatrix}$$

Suppose that the collected U_{nm} values result in actual CPU percentages $X_m(T_k)$ equal to $\{11\%, 8\%, 19\%, 11\%, 19\%\}$. The tolerances for thresholding are set at 2%, so the algorithm calculates group $A = \{1,2\}$ and group $B = \{3,5\}$. Therefore, we must activate domains 1 & 2 and deactivate domains 3 & 5. Now based on the algorithm described (SDA), there is no instances of domains 1 and 2 activated in appliance 4, so there is no need to deactivate instances of domains 3 & 5 in that appliance. However, as there are instances of domains 1 and 2 running in appliances 1, 2 and 3, there will be deactivations of domains 3 and 5 in these appliances. Note that, because there is only one instance of domain 3 activated in appliance 2 and only one instance of domain 5 activated in appliance 3, these two entries will be kept unchanged. Because of the deactivation, as some of the CPU resource utilized by domain 3 and 5 is freed up, under-utilized domain 1 and 2 can take advantage of that and activate one more instance of domain 1 and 2 in appliance 2, domain 1 in appliance 1 (domain 2 cannot be activated in appliance 1 as it is not already activated there) and domain 2 in appliance 3. So, after SDA, we will get (changed values are in bold face),

$$B = \begin{bmatrix} 1 & 0 & \mathbf{9} & 1 & \mathbf{3} \\ 10 & 3 & 1 & 2 & \mathbf{3} \\ 0 & 5 & \mathbf{7} & 4 & 1 \\ 0 & 0 & 2 & 4 & 10 \end{bmatrix}$$

and after SAA, we will get instantiation matrix as follows (changed values are in bold face),

$$B = \begin{bmatrix} \mathbf{2} & 0 & 9 & 1 & 3 \\ \mathbf{11} & \mathbf{4} & 1 & 2 & 3 \\ 0 & \mathbf{6} & 7 & 4 & 1 \\ 0 & 0 & 2 & 4 & 10 \end{bmatrix}$$

4 SIMULATION AND ANALYSIS

4.1 Simulation Goals and Assumptions

Despite the strong engineering intuition, we have no theoretical proof that the SDA/SAA algorithm will be able to satisfy any arbitrary, desired values of CPU allocations. Therefore, in order to verify the proposed algorithm, we evaluated the multi-service multi-appliance system by developing a discrete-event simulator in C. We focused our analysis in this paper on the following three sets of questions:

Q1. Does SDA/SAA "work" (i.e., can it meet the P_m service differentiation goals?) Figures 4 and 5 are representative results in this regard.

Q2. Is SDA/SAA indeed "better" than the other open-loop, static approaches (i.e., does it have the advantages described in section 2.1)? Figure 6 (partly) answers this question.

Q3. How do algorithm parameters (i.e., UT_m/DT_m, N, M, $\{P_m\}$, initial B_{nm} values) affect the behavior of SDA/SAA? Figures 7 through 12 partly answer this question.

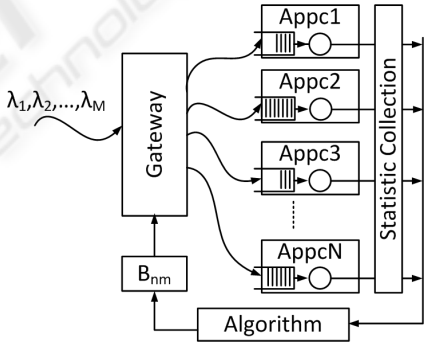

Figure 3: Simulator Design

The simulation model is depicted in Figure 3. The service requests arrive at the system in a random fashion. The gateway arrival process for service domain m is modeled for simplicity as a Poisson process[3] with arrival rate λ_m. The CPU service time for requests from domain m is a uniform random variable with average value ES_m. For simplicity, all appliances are considered homogeneous. They employ a single, infinite-capacity FIFO buffer for all domains activated in them; their CPU capacity is normalized to 1 unit. Therefore, the CPU utilization of (and thus the CPU allocation to) a service domain m would be $\lambda_m \cdot ES_m$.

[3]Since the system is controlled in a closed-loop fashion, the nature of the randomness in the arrival (and service time process) is not that critical.

4.2 Simulation Results and Analysis

Due to lack of space, in this paper we only include representative results. A more comprehensive set of results and analysis (including confidence intervals) are provided in (Habib, 2009).

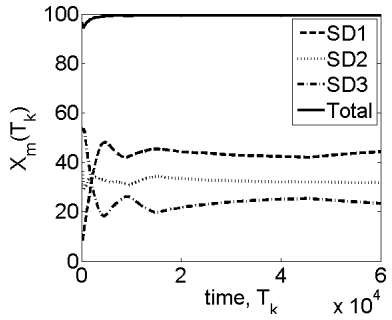

Figure 4: Utilization $X_m(T_k)$ vs time.

To answer question **Q1**, we varied the number of appliances, N from 1 to 10; the number of service domains, M from 1 to 20. For the results depicted in Fig. 4, we set $N = 4$, $M = 3$, the desired goals are $\{P_m\} = \{44\%, 33\%, 22\%\}$ with 2% up and down threshold tolerances. All domains have the same service times and arrival rates. We initialized the instantiation matrix to the same values in all four appliances; in order to create an "unfavorable" situation for the algorithm, the number of instances initialized were $\{2, 5, 10\}$ for the three domains respectively, as opposed to the desired ratios of 44/33/22 respectively. Fig. 4 shows that the SDA/SAA algorithm meets the desired goal despite the unfavorable initial instantiation in the entire cluster. In this simulation, the total arrival rate was chosen high enough to keep the CPUs busy, hence the total utilization (also shown in Fig. 4) approaches 100%.

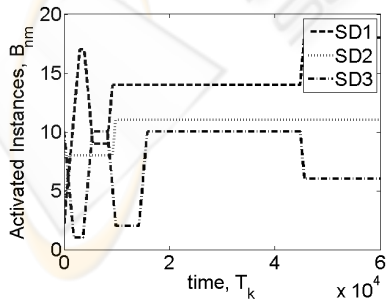

Figure 5: Variation in B_{nm} values, appliance 1.

In Fig. 5, we observe how the algorithm alters the number of instances of service domains in the appliances to achieve the goal. In all figures that depict activated instances, values for appliance 1 are shown (graphs are similar for other appliances). The algorithm causes oscillation in the beginning as for lower value of k, $X_m(k)$ changes abruptly which in turn causes oscillations in the values of B_{mn}.

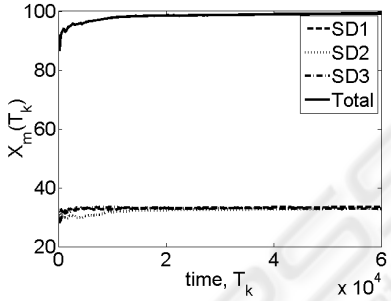

Figure 6: Utilization $X_m(T_k)$ vs time.

We demonstrate advantages (c) and (d) mentioned in section 2.1 in detail in (Habib, 2009). In this paper, to answer question **Q2**, observe that desired P_m goals depend heavily on the actual arrival rates (which may not be known in a real system). For example, suppose we specify $\{P_1, P_2, P_3\} = \{44\%, 33\%, 22\%\}$ and the arrival rates and the average service times for the three service domains are equal. A static allocation, in this case, would allocate CPU times in the ratios 44% : 33% : 22%, wasting 11% for SD1, depriving SD3 of 33-22=11% and leaving a 22% "white space" (unused CPU resource). Figure 6 shows how SDA/SAA could achieve an equal allocation of CPU resources in this scenario, with a total CPU allocation of 100%, which would eliminate the white space altogether.

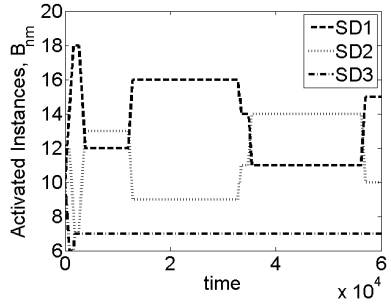

Figure 7: Variation in B_{nm} Values, in Appliance 1, for $N = 1$ Appliances.

To answer question **Q3** involves varying the algorithm parameters N, M, UT_m/DT_m, P_m, initial B_{nm}. In all our experiments, the behavior of the algorithm (i.e., the nature of variations in the B_{nm} values) as well as its performance (i.e., the achieved percentages) did

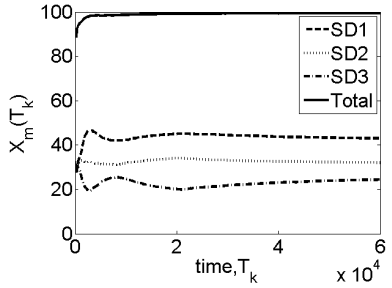

Figure 8: Utilization $X_m(T_k)$ vs time, in Appliance 1, for $N = 10$ Appliances.

not change as we varied the number of appliances N or the number of service domains M. In the interest of saving space, we show in figures 7 and 8 some results only for the "boundary cases" $N = 1$ and $N = 10$ we tried. The experiments had the same setting as the one used in question **Q1**. As expected, the results agree with those depicted in Fig. 4.

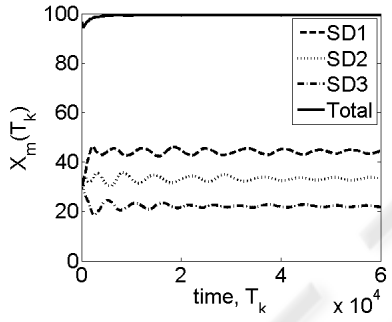

Figure 9: Utilization $X_m(T_k)$, effect of strict tolerances.

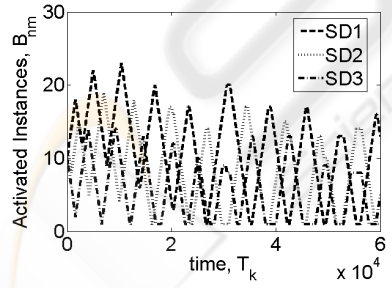

Figure 10: Variation in B_{nm}, effect of strict tolerances.

The effect of the tolerance parameters UT_m/DT_m is typical of the "oscillatory" behavior depicted in Fig. 10. The figure was produced with (a rather strict) setting of $UT_m = DT_m = 0.1\%$ for all domains; the rest of the experiment setup is the same as the one depicted in the scenario of question Q1. In general, stricter tolerances cause more oscillations in both the goals and the instantiation matrix values (compare Fig. 9 to Fig. 4 and Fig. 10 to Fig. 5). Throughout the experiment, an initial value of $B_{nm} = 10, \forall n,m$ was used.

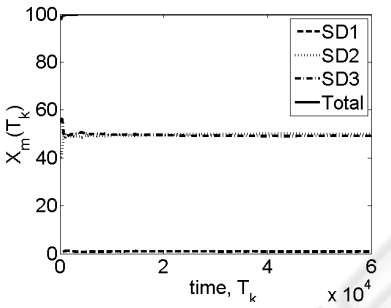

Figure 11: Utilization $X_m(T_k)$, "non-achievable" P_m goals.

In general, the P_m parameter can be set by the system administrator in one of two possible ways: "achievable" or "non-achievable". In the first, the arrival rate λ_m and average service times ES_m of the domain are such that $\lambda_m \cdot ES_m \geq P_m$; in other words, there is enough traffic to take advantage of the allocated CPU resource. Figure 4 is an example of this case. In the second, we have that $\lambda_m \cdot ES_m < P_m$; in this case, the domain does not have enough traffic to take advantage of the allocated CPU resource. As with all feedback-based algorithms, this situation may "mislead" the algorithm into always activating additional instances of the domain, causing "instability" and eventually affecting other domains too[4].

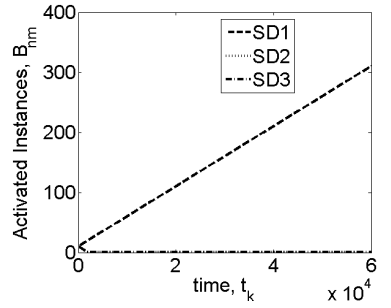

Figure 12: Potential for instability, "non-achievable" P_m goals.

Figure 11 exemplifies what can happen when "non-achievable" goals are set. In this experiment, we set again $\{P_1, P_2, P_3\} = \{44\%, 33\%, 22\%\}$. The arrival rate for SD1 was set low, so that this domain would never reach a 44% CPU utilization, even if it was given full access of the CPUs; its maximum utilization will eventually approach $\lambda_1 \cdot ES_1 \approx 6\%$ in this

[4]This is one of the "special cases" we alluded to in section 3.3.

experiment. The other two domains produced enough traffic to fully utilize their desired percentages. As Figure 11 shows, these two domains (over)achieve their desired percentages. Figure 12 explains why. The algorithm keeps activating instances for SD1, the "underachieving" domain, at the expense of the other two domains, which are left with only one activated instance each; this explains why these two domains get an equal share of the CPU. The total CPU utilization stays at 100%, as shown in Fig. 11, eliminating any white space.

5 CONCLUSIONS

In this paper, we proposed SAA/SDA algorithm, a closed-loop, feedback-based algorithm that provides service differentiation based on CPU utilization measurements in a cluster of middleware appliances. The appliances employ FIFO buffering and thus differentiation is controlled by activation/deactivation of service domains. The algorithm achieves the differentiation goals by controlling the rate at which service requests are sent to individual appliances in the cluster; it does not rely on a priori knowledge of service domain statistics. It has the following advantages: (a) it is capable of providing arbitrary allocation of CPU resources to service domains, thus achieving true service differentiation, (b) it utilizes appliance resources in an efficient manner, and thus it leverages processing white-space across all appliances, (c) it increases service locality, and, (d) it does not require manual configurations. We have demonstrated such advantages with extensive simulations.

REFERENCES

Chandrashekar, J., li Zhang, Z., Duan, Z., and Hou, Y. T. (2003). Service oriented internet. In *Proceedings of the 1st ICSOC*.

Erl, T. (2004). *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR.

Habib, M. (2009). Provisioning algorithms for service differentiation in middleware appliance clusters. Master's thesis, North Carolina State University.

Menascé, D. A., Barbará, D., and Dodge, R. (2001). Preserving qos of e-commerce sites through self-tuning: a performance model approach. In *EC '01: Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 224–234, New York, NY, USA. ACM.

Michael Huhns, M. P. S. (2005). Service oriented computing: Key concepts and principle. *IEEE Internet Computing, IEEE Computer Society*, pages 75–82.

Parekh, A. K. and Gallager, R. G. (1993). A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357.

Ranjan, S., Rolia, J., Fu, H., and Knightly, E. (2002). Qos-driven server migration for internet data centers. In *Proc. Tenth IEEE International Workshop on Quality of Service*, pages 3–12.

Sharma, A., Adarkar, H., and Sengupta, S. (2003). Managing qos through prioritization in web services. In *Proceedings on Fourth International Conference on Web Information Systems*, pages 140–148.

Wang, X., Du, Z., Chen, Y., Li, S., Lan, D., Wang, G., and Chen, Y. (2008). An autonomic provisioning framework for outsourcing data center based on virtual appliances. *Cluster Computing*, 11(3):229–245.

Zhang, C., Chang, R. N., Perng, C.-S., So, E., Tang, C., and Tao, T. (2008). Leveraging service composition relationship to improve cpu demand estimation in soa environments. In *SCC '08: Proceedings of the 2008 IEEE International Conference on Services Computing*, pages 317–324, Washington, DC, USA. IEEE Computer Society.

Zhu, H., Tang, H., and Yang, T. (2001). Demand-driven service differentiation for cluster-based network servers. In *In Proc. IEEE INFOCOM*, pages 679–688.