A Self-Tuning Fuzzy Control Approach for End-to-End QoS Guarantees in Web Servers*

Jianbin Wei and Cheng-Zhong Xu

Department of Electrical and Computer Engineering
Wayne State University, Detroit, Michigan 48202
Email: {jbwei, czxu}@wayne.edu

Abstract. It is important to guarantee end-to-end quality of service (QoS) under heavy-load conditions. Existing work focus on server-side request processing time or queueing delays in the network core. In this paper, we propose a novel framework *e*QoS to monitoring and controlling client-perceived response time in Web servers. The response time is measured with respect to requests for Web pages that contain multiple embedded objects. Within the framework, we propose an adaptive fuzzy controller, STFC, to allocating server resources. It deals with the effect of process delay in resource allocation by its two-level self-tuning capabilities. Experimental results on PlanetLab and simulated networks demonstrate the effectiveness of the framework: it controls client-perceived pageview response time to be within 20% of a pre-defined target. In comparison with static fuzzy controller, experimental results show that, although the STFC has slightly worse performance in the environment where the static fuzzy controller is best tuned, because of its self-tuning capabilities, it works better in all other test cases by 25% in terms of the deviation from the target response time. In addition, due to its model independence, the STFC outperforms the linear proportional integral (PI) and adaptive PI controllers by 50% and 75%, respectively.

1 Introduction

In the past decade we have seen an increasing demand for provisioning of quality of service (QoS) guarantees to various network applications and clients. There existed many work on provisioning of QoS guarantees. Most of them focused on Web servers without considering network delays [1, 6], on individual network router [7], or on clients with assumptions of QoS supports in networks [8]. Recent work [11] on end-to-end QoS guarantees in network cores aimed to guarantee QoS measured from server-side network edges to client-side network edges without considering delays incurred in servers.

In practice, client-perceived QoS is attributed by network delays and by server-side request queueing delays and processing time. The objective of this paper is to guarantee end-to-end QoS in Web servers. To provide such QoS guarantees, service quality must be accurately measured in real time so that server resources can be allocated promptly. Most recently, the ksniffer approach presented in [14] realized such on-line real-time measurement and made such guarantees possible.

* This work was supported in part by US NSF grant ACI-0203592 and NASA grant 03-OBPR-01-0049.

The first contribution in this paper is a novel *e*QoS framework to monitoring and controlling client-perceived QoS in Web servers. To the best of our knowledge, the *e*QoS is the first one to guarantee client-perceived end-to-end QoS based on the ksniffer ideas of real-time QoS measurement. Moreover, because more than 50% of Web pages have one or more embedded objects [9], the guaranteed QoS is measured with respect to requests for *whole* Web pages that contain multiple embedded objects, instead of requests for a single object [1, 2] or connection delay in Web servers [19].

The second contribution of this paper is a two-level self-tuning fuzzy controller (STFC) that requires no accurate server model to allocate server resources within the *e*QoS. Traditional linear feedback control has been applied as an analytic method for QoS guarantees in Web servers because of its self-correcting and self-stabilizing behavior [1]. It adjusts the allocated resource of a client class according to the difference between the target QoS and the achieved one in previous scheduling epochs. It is well known that linear approximation of a nonlinear system is accurate only within the neighborhood of the point where it is linearized. In fast changing Web servers, the operating point changes dynamically and simple linearization is inappropriate.

In Web servers, resource allocation must be based on accurately measured effect of previous resource allocation on the client-perceived response time of Web pages. According to the HTTP protocol, on receiving a request for the base (or container) of a Web page, the server needs to schedule the request according to its resource allocation. At this point, it is impossible to measure the client-perceived response time of the Web page because the server needs to handle the request and the response needs to be transmitted over the networks. An accurate measurement of resource-allocation effect on response time is thus delayed. Consequently, the resource allocation is significantly complicated because it has to be based on an inaccurate measurement. We refer to the latency between allocating server resources and accurately measuring the effect of the resource allocation on provided service quality as *process delay*.

The STFC overcome the existing approaches' limitations with its two-level self-tuning capability. On its first level is a resource controller that takes advantage of fuzzy control theory to address the issue of lacking accurate server model due to the dynamics and unpredictability of pageview request traffic. On the second level is a scaling-factor controller. It aims to compensate the effect of the process delay by adjusting the resource controller's output scaling factor according to transient server behaviors. We note that fuzzy control theory was recently used by others for QoS guarantees as well [12, 16]. Their approaches, however, are non-adaptive. They cannot guarantee client-perceived pageview response time in the presence of the process delay.

We implement a prototype of *e*QoS in Linux. We conduct experiments across wide-range server workload conditions on PlanetLab test bed [18]. and on simulated networks. The experimental results demonstrate that provisioning of client-perceived QoS guarantees is feasible and the *e*QoS is effective in such provisioning: it controls the deviation of client-perceived average pageview response time to be within 20% of a pre-defined target. For comparison, we also implement three other controllers within the *e*QoS framework: a static fuzzy that uses similar as the one in [12], a linear PI controller, and an adaptive PI controller that bears resemblance to the approach in [10]. Experimental results show that, although the STFC works slightly worse than the non-

adaptive fuzzy controller in the environment where the non-adaptive fuzzy controller is best tuned, because of its self-tuning capabilities, it has better performance in all other test cases by 25% in terms of the deviation from the target response time. The STFC outperforms the linear PI and adaptive PI controllers by 50% and 75%, respectively.

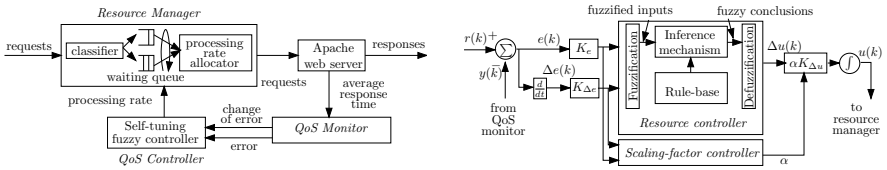
The structure of the paper is as follows. Section 2 presents the *e*QoS framework. Section 3 presents the two-level STFC. Section 4 evaluates the performance of the *e*QoS in real-world and simulated networks and compares the performance between different controllers. Section 5 reviews related work and Section 6 concludes the paper.

2 The *e*QoS Framework

The *e*QoS framework is designed to guarantee the average pageview response time of premium clients $W(k)$ to be close to a target $D(k)$ in heavy-loaded servers. Because the server load can grow arbitrary high, it is impossible to guarantee QoS of all clients under heavy-load conditions. Client-perceived response time is the time interval that starts when a client sends the first request for the Web page to the server and ends when the client receives the last object of the Web page. In this work we use the Apache Web server with support of HTTP/1.1. We assume that all objects reside in the same server so that we can control the processing of the whole Web page. The latency incurred in resolving domain name into IP address is not considered because it is normally negligible. As shown in [15], 90% of the name-lookup requests have response time less than 100 *ms* for all of their examined domain name servers except one.

The *e*QoS framework consists of four components: a Web server, a QoS controller, a resource manager, and a QoS monitor. Fig. 1(a) illustrates the components and their interactions. The QoS controller determines the amount of resource allocated to each class. It can be any controller designed for the provisioning of QoS guarantees. In addition to the STFC, current implementation includes three other controllers: a non-adaptive fuzzy, a PI, and an adaptive PI controllers for comparison. The QoS monitor measures client-perceived pageview response time using ideas similar as those presented in [14].

The resource manager classifies and manages client requests and realizes resource allocation between classes. It comprises of a classifier, several waiting queues, and a processing-rate allocator. The classifier categorizes a request's class according to rules defined by service providers. The rules can be based on the request's header information (e.g., IP address and port number). Without the *e*QoS, a single waiting queue is created in the kernel to store all client requests for each socket. In the *e*QoS, requests are stored in their corresponding waiting queues in the resource manager. The requests from the same class are served in first-come-first-served manner. The process-rate allocator realizes resource allocation between different classes. Since every child process in the Apache Web server is identical, we realize the processing-rate allocation by controlling the number of child processes that a class is allocated. In addition, when a Web server becomes overloaded, admission control mechanisms [25] can be integrated into the resource manager to ensure the server's aggregate performance.



(a) Implementation structure of the *e*QoS (b) The structure of the STFC

Fig. 1. The structure of the *e*QoS framework.

3 The Self-Tuning Fuzzy Controller

To guarantee client-perceived QoS effectively, the QoS controller must address issues of the process delay in resource allocation without any assumption of pageview request traffic model. To the end, we present a self-tuning fuzzy controller, STFC. Fig. 1(b) presents its structure.

3.1 The Resource Controller

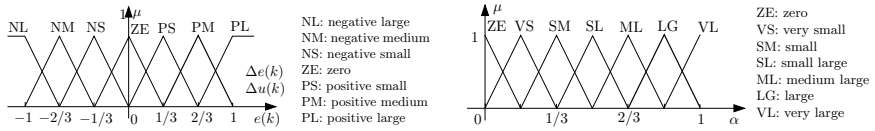
As shown in Fig. 1(b), the resource controller consists of four components. The rule-base contains a set of *If-Then* rules about quantified control knowledge about how to adjust the resource allocated to premium class according to $e(k)$ and $\Delta e(k)$ in order to provide QoS guarantees. The fuzzification interface converts controller inputs into certainties in numeric values of the input membership functions. The inference mechanism activates and applies rules according to fuzzified inputs, and generates fuzzy conclusions for defuzzification interface. The defuzzification interface converts fuzzy conclusions into the change of resource of premium class in numeric value.

The resource controller presented in Fig. 1(b) also contains three scaling factors: input factors K_e and $K_{\Delta e}$ and output factor $\alpha K_{\Delta u}$. They are used to tune the controller's performance. The actual inputs of the controller are $K_e e(k)$ and $K_{\Delta e}\Delta e(k)$. In the output factor, α is adjusted by the scaling-factor controller. Thus, the resource allocated to premium class during the $(k+1)$th sampling period $u(k+1)$ is $\int \alpha K_{\Delta u}\Delta u(k)dk$.

The parameters of the control loop as shown in Fig. 1(b) are defined as follows. The reference input for kth sampling period $r(k)$ is $D(k)$. The output of the loop is the achieved response time $W(k)$. The error $e(K)$ and the change of error $\Delta e(k)$ are defined as $D(k) - W(k)$ and $e(k) - e(k-1)$, respectively.

It is well known that the bottleneck resource plays an important role in determining the service quality a class receives. Thus, by adjusting the bottleneck resource a class is allocated, we are able to control its QoS: The more resource it receives, the smaller response time it experiences. The key challenge in designing the resource controller is translating heuristic control knowledge into a set of control rules so as to provide QoS guarantees without an accurate model of continuously changing Web servers.

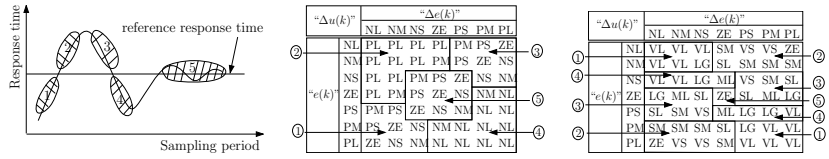
In the resource controller, we define the control rules using linguistic variables. For brevity, linguistic variables "$e(k)$", "$\Delta e(k)$", and "$\Delta u(k)$" are used to describe $e(k)$, $\Delta e(k)$, and $\Delta u(k)$, respectively. The linguistic variables assume linguistic values $NL, NM, NS, ZE, PS, PM, PL$. Their meanings are shown in Fig. 2(a).



(a) Membership functions of e, Δe, and Δu (b) The membership functions of α

Fig. 2. The membership functions of the STFC.

We next analyze the effect of the controller on the provided services as shown in Fig. 3(a). In this figure, five zones with different characteristics can be identified. Zone 1 and 3 are characterized with opposite signs of $e(k)$ and $\Delta e(k)$. That is, in zone 1, $e(k)$ is positive and $\Delta e(k)$ is negative; in zone 3, $e(k)$ is negative and $\Delta e(k)$ is positive. In these two zones, it can be observed that the error is self-correcting and the achieved value is moving towards to the reference value. Thus, $\Delta u(k)$ needs to set either to speed up or to slow down current trend. Zone 2 and 4 are characterized with the same signs of $e(k)$ and $\Delta e(k)$. That is, in zone 2, $e(k)$ is negative and $\Delta e(k)$ is negative; in zone 4, $e(k)$ is positive and $\Delta e(k)$ is positive. Different from zone 1 and zone 3, in these two zones, the error is not self-correcting and the achieved value is moving away from the reference value. Therefore, $\Delta u(k)$ should be set to reverse current trend. Zone 5 is characterized with rather small magnitudes of $e(k)$ and $\Delta e(k)$. Therefore, the system is at a steady state and $\Delta u(k)$ should be set to maintain current state and correct small deviations from the reference value.



(a) Illustration of control effect (b) Rule-base of the resource controller (c) Rule-base of the scaling-factor controller

Fig. 3. Fuzzy control rules in the STFC.

By identifying these five zones, we design the fuzzy control rules as summarized in Fig. 3(b). A general linguistic form of these rules is read as: *If* premise *Then* consequent. Let $rule(m, n)$, where m and n assume linguistic values, denote the rule of the (m, n) position in Fig. 3(b). As an example, $rule(PS, PM) = NL$ reads that: *If* the error is positive small *and* the change of error is positive medium *Then* the change of resource is negative large. Note that the control rules are designed based on the analysis of resource-allocation on achieved response time. It avoids the needs of an accurate server model.

In the resource controller, the meaning of the linguistic values is quantified using "triangle" membership functions, which are most widely used in practice, as shown in Fig. 2(a). In Fig. 2(a), the x-axis can be $e(k)$, $\Delta e(k)$, or $\Delta u(k)$. The mth membership function quantifies the *certainty* (between 0 and 1) that an input can be classified as

linguistic value m. The fuzzification component translates the inputs into corresponding certainty in numeric values of the membership functions. The inference mechanism is to determine which rules should be activated and what conclusions can be reached. Based on the outputs of the inference mechanism, the defuzzification component calculates the fuzzy controller output, which is a combination of multiple control rules, using "center average" method.

3.2 The Scaling-factor Controller

To successfully design the resource controller discussed in Section 3.1, the effect of the process delay must be compensated. To the end, we design a scaling-factor controller to adaptively adjust $\alpha K_{\Delta u}$ according to the transient behaviors of a Web server in a way similar to [13]. The selection of output scaling factor $\alpha K_{\Delta u}$ is because of its global effect on the control performance.

The scaling-factor controller consists of the same components as the resource controller. The membership functions of "α" (the corresponding linguistic variable of α) also have "triangle" shape as shown in Fig. 2(b). Because α needs to be positive to ensure the stability of the control system, "α" assumes different linguistic values from "$e(k)$" and "$\Delta e(k)$". Fig. 2(b) also shows the linguistic values and their meanings.

The control rules of the scaling-factor controller are summarized in Fig. 3(c) with following five zones.

1. When $e(k)$ is large but $\Delta e(k)$ and $e(k)$ have the same signs, the client-perceived response time is not only far away from the reference value but also it is moving farther away. Thus, α should be set large to prevent the situation from further worsening.
2. When $e(k)$ is large and $\Delta e(k)$ and $e(k)$ have the opposite signs, α should be set at a small value to ensure a small overshoot and to reduce the settling time without at the cost of responsiveness.
3. When $e(k)$ is small, α should be set according to current server states to avoid large overshoot or undershoot. For example, when $\Delta e(k)$ is negative large, a large α is needed to prevent the upward motion more severely and can result in a small overshoot. Similarly, when $e(k)$ is positive small and $\Delta e(k)$ is negative small, then α should be very small. The large variation of α is important to prevent excessive oscillation and to increase the convergence rate of achieved service quality.
4. The scaling-factor controller also provides regulation against the disturbances. When a workload disturbance happens, $e(k)$ is small and $\Delta e(k)$ is normally large with the same sign as $e(k)$. To compensate such workload disturbance, α is set large.
5. When both $e(k)$ and $\Delta e(k)$ are very small, α should be around zero to avoid chattering problem around the reference value.

The operation of the STFC has two steps. First, we tune the K_e, $K_{\Delta e}$, and $K_{\Delta u}$ through trials and errors. In the step the scaling-factor controller is off and α is set to 1. In the second step, the STFC is turned on to control resource allocation in running Web servers. The scaling-factor controller is on to tune α adaptively. The K_e and $K_{\Delta e}$ are kept unchanged and the $K_{\Delta u}$ is set to three times larger than the one obtained in previous step to maintain the responsiveness of the STFC during workload disturbances.

Finally we remark that the STFC has small overhead because at most eight rules are on at any time in the STFC. In addition, the controller only needs to adjust resource allocation once a sampling period. We conducted experiments with STFC-on and STFC-off and observe their performance difference is within 1%. Furthermore, the implementation of the STFC totaled less than 100 lines of C code.

4 Performance Evaluations

We define a metric of relative deviation $R(e)$ to measure the performance of the *e*QoS:

$$R(e) = \frac{\sqrt{\sum_{k=1}^{n} \left(D(k) - W(k)\right)^2 / n}}{D(k)} = \frac{\sqrt{\sum_{k=1}^{n} e(k)^2 / n}}{D(k)}. \quad (1)$$

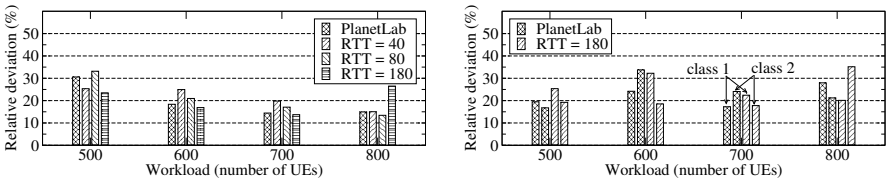
The smaller the $R(e)$, the better the controller's performance. We have conducted experiments on the PlanetLab test bed to evaluate the performance of the *e*QoS in a real-world environment. The clients reside on 9 geographically diverse nodes: Cambridge in Massachusetts, San Diego in California, and Cambridge in the United Kingdom. We assume that premium and basic clients are from all these nodes for fairness between clients with different network connections. The Web server is a Dell PowerEdge 2450 configured with dual-processor (1 GHz Pentium III) and 512 MB main memory and is located in Detroit, Michigan. During the experiments, the RTTs between the server and the clients are around 45 *ms* (Cambridge), 70 *ms* (San Diego), and 130 *ms* (the UK).

The server workload was generated by SURGE [3]. In the emulated Web objects, the maximum number of embedded objects in a given page was 150 and the percentage of base, embedded, and loner objects were 30%, 38%, and 32%, respectively. The Apache Web server was used to provide Web services with support of HTTP/1.1. The number of the maximal concurrent child processes was set to 128. In the experiments with two classes, we aimed to keep the average response time of premium class to be around 5 seconds. In the experiments with three classes, we assumed the target of class 1 was 5 seconds and that of class 2 was 11 seconds because they are rated as "good" and "average", respectively [5]. We aimed to provide guaranteed service when the number of UEs was between 500 and 800. When the number of UEs is less than 500, the average response time of all Web pages is around 5 seconds. When the number of UEs is larger than 800, we have observed refused connections using unmodified Apache Web server and admission control mechanisms should be employed.

To investigate the effect of network latency on the performance of the *e*QoS, we have implemented a network-delay simulator in a similar way to [22]. With the RTT set as 180 *ms*, ping times were showing a round trip of around 182 *ms* using the simulator. In the experiments on the simulated networks, the RTT between clients and servers was set to be 40, 80, or 180 *ms* that represent the latency within the continental U.S., the latency between the east and west coasts of the U.S., and the one between the U.S. and Europe, respectively [20].

4.1 Effectiveness of the *e*QoS

To evaluate the effectiveness of the *e*QoS, we have conducted experiments under different workloads and network delays with two and three client classes. In the experiments, the system was first warmed up for 60 seconds and then the controller was on. The size of sampling period was set to 4 seconds. The effect of the sampling period on the performance of the *e*QoS shall be discussed in Section 4.3. Fig. 4 presents the experimental results. Fig. 4(a) shows the relative deviations of the premium class relative to the reference value (5 seconds). From the figure we observe that all the relative deviations are smaller than 35%. Meanwhile, most of them are around 20%. It means the size of deviations is normally around 1.0 seconds. Fig. 4(b) presents the results with three classes. Because we observe no qualitative differences between the results with different RTTs in the simulated networks, we only present the results where RTT was set to 180 *ms* for brevity. From the figure we see that most of the relative deviations are between 15% and 30%. These demonstrate the effectiveness of the *e*QoS.

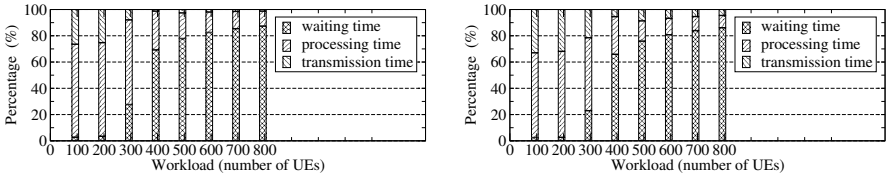


(a) Two classes (b) Three classes

Fig. 4. The performance of the *e*QoS with two and three classes.

We then investigate why it is feasible to guarantee end-to-end QoS from server side under heavy-load conditions. The pageview response time consists of server-side waiting and processing time and network transmission time. The waiting time is the time interval that starts when a connection is accepted by the operating system and ends when the connection is passed to the Apache Web server to be processed. The processing time is the time that the Apache Web server spends on processing the requests for the whole Web page, including the base HTML file and its embedded objects. The transmission time includes the complete transfer time of client requests and all server responses over the networks. We instrumented the Apache Web server to record the processing time.

Fig. 5 shows the breakdown of response time. From the figure we observe that, when the number of UEs is larger than 400, the server-side waiting time is the dominant part of client-perceived response time. This finding is consistent with those in [4]. It is because that, when the server is heavily loaded, the child processes of the Apache Web server are busy in processing accepted client requests. The newly incoming client requests then have to wait. Furthermore, we also observe that the transmission time is only a small part of response time when server workload is high. It indicates that, although the service providers have no control over the network transmissions, they can still control the client-perceived response time by controlling the server-side waiting time and processing time.

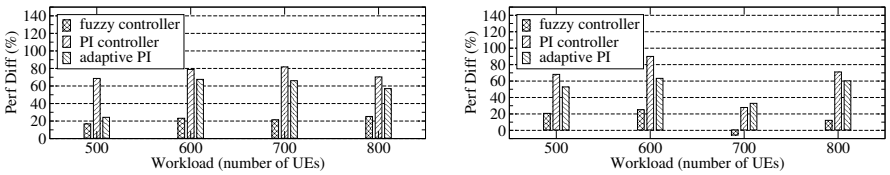


(a) RTT = 40 *ms* (b) RTT = 180 *ms*

Fig. 5. The breakdown of response time of Web pages under different RTTs.

4.2 Comparison with Other Controllers

Within the *e*QoS framework, we also implement three other controllers: a fuzzy controller without self-tuning, a traditional PI controller, and an adaptive PI controller using the basic idea of [10]. We have specifically tuned the fuzzy and the PI controllers in an environment where the number of UEs was set to 700 and RTT was set to be 180 *ms* on the simulated networks. We define the performance difference $PerDiff$ between the STFC and other controller as $(R(e)_{other} - R(e)_{STFC})/R(e)_{STFC}$. The $R(e)_{other}$ and $R(e)_{STFC}$ are the relative deviations of other controller and the STFC, respectively. Fig. 6 presents the performance difference due to compared controllers.



(a) PlanetLab (b) RTT = 180 *ms*

Fig. 6. The performance comparison in PlanetLab and simulated networks.

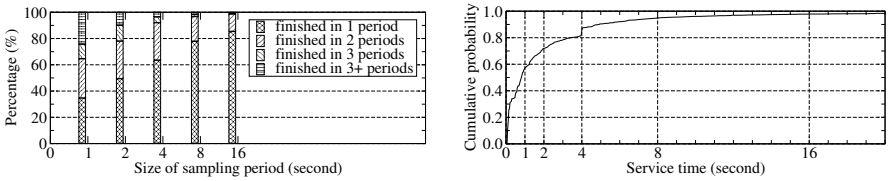
From Fig. 6(b) we observe that the STFC provides worse services than the non-adaptive fuzzy controller when the number of UEs is 700 and the RTT is 180 *ms*. The behavior is expected because a self-tuning controller cannot provide better performance than a non-adaptive controller that has been specifically tuned for a certain environment. Even under such environment, the performance difference is only -6%. Under all other conditions, the STFC provides 25% better services than the non-adaptive fuzzy controller in terms of performance difference because the STFC further adjusts $\alpha K_{\Delta u}$ adaptively according to the transient behaviors of the Web server. Such tuning is important to compensate the effect of the process delay in resource allocation.

In comparison with the PI controller, the STFC achieves better performance even when the PI controller operates under its specifically tuned environment, which can be observed in Fig. 6(b). When the number of UEs is 700 and RTT is 180 *ms*, their performance difference is 28%. From Fig. 6 we observe that all performance differences of the PI controller are larger than 60% and the average is around 75%. The poor perfor-

mance of the PI controller is due to its inaccurate underlying model. In the PI controller, we follow the approach in [10] and model the server as an $M/GI/1$ processor sharing system. It is known that the exponential inter-arrival distribution is unable to characterize the Web server [17]. Thus, the model is inaccurate. Similarly, although the adaptive PI improves upon the non-adaptive PI controller, it still has worse performance than the STFC and the fuzzy controller. Its average performance difference in relation to the STFC is around 50%. The poor performance of these two controllers is because they provide no means to compensate the effect of the process delay.

4.3 The Process Delay in Resource Allocation

Aforementioned, the process delay in resource allocation affects the performance of a controller. We have conducted experiments to quantify it. For brevity, we only present the results on simulated networks where the number of UEs was 700 and the RTT was 180 *ms*. Fig. 7(a) shows the percentage of requests finished within different numbers of sampling period after being admitted. Fig. 7(b) depicts corresponding cumulative distribution function of the service time, which is the time the Web server spends in processing requests. Comparing Fig. 7(a) and Fig. 7(b) we observe that, although over 95% of the requests are finished in 8 seconds after being admitted, only 77.8% of them are processed within the same sampling period when it is set to 8 seconds. Moreover, it also indicates that 22.2% of the measured response time are affected by the resource allocation performed more than one sampling periods ago. Consequently, the resource-allocation effect cannot be accurately measured promptly.



(a) The percentage of processed requests as a function of the size of sampling period

(b) The cumulative distribution function of service time

Fig. 7. The process delay in resource allocation.

To provide QoS guarantees, however, the resource allocation in Web servers should be based on an accurately measured effect of previous resource allocation on client-perceived QoS. It in turn controls the order in which client requests are scheduled and processed. The existing process delay has been recognized as one of the most difficult dynamic element naturally occurring in physical systems to deal with [21]. It sets a fundamental limit on how well a controller can fulfill design specifications because it limits how fast a controller can react to disturbances.

The process delay also affects the selection of an appropriate sampling period. Due to space limitation, we summerize our observation. From the results, we observe that

the deviation decreases with the increase of the sampling period. It is because that the measured effect of resource allocation is more accurate using a large sampling period than a small one. When the sampling-period size continues to increase, the relative deviation turns to increases. It is because that, with the increase of sampling period, the processing rate of premium class is adjusted less frequently. Consequently, the *e*QoS becomes less adaptive to the transient workload disturbances. Based on the results, we set the size of sampling period to 4 seconds.

5 Related Work

Early work focused on providing differentiated services to different client classes using priority-based scheduling [2]. Although they are effective in providing differentiated services, they cannot guarantee the QoS a class received. To guarantee the QoS of a class, queueing-theoretic approaches have been proposed. The performance highly depends on the parameter estimation, such as the traffic variance, which is difficult to be accurate. To reduce the variance of achieved QoS, traditional linear feedback control has also been adapted [1, 24]. Because the behavior of a Web server changes continuously,, the performance of the linear feedback control is limited. In comparison, our approach takes advantage of fuzzy control theory to manage the server-resource allocation.

Recent work have applied adaptive control [10] and machine-learning [23] to address the lack of accurate server model. Although these approaches provide better performance than non-adaptive linear feedback control approaches under workload disturbances, the ignorance of the process delay limits their performance. Fuzzy control theory has also been applied in providing QoS guarantees [12, 16]. The objective of the STFC is different in that its focus is on providing end-to-end QoS guarantees. Moreover, the STFC explicitly addresses the inherent process delay in resource allocation.

6 Conclusions

In the paper, we have proposed a novel framework *e*QoS to providing end-to-end pageview response time guarantees. Within the framework, we have proposed a two-level self-tuning fuzzy controller, which does not require accurate server model, to explicitly addressing the process delay in resource allocation. The experimental results on PlanetLab and simulated networks have shown that it is effective in providing such QoS guarantees. They also demonstrated the superiority of the STFC over other controllers with much smaller deviations.

References

1. T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, January 2002.
2. J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in Web content hosting. In *Proceedings of ACM SIGMETRICS Workshop on Internet Server Performance*, 1998.

3. P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS*, 1998.
4. P. Barford and M. Crovella. Critical path analysis of TCP transactions. *IEEE/ACM Transactions on Networking*, 9(3):238–248, 2001.
5. N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into Web server design. In *Proceedings of WWW*, 2000.
6. N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, 1999.
7. C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional differentiated services: Delay differentiation and packet scheduling. *IEEE/ACM Transactions on Networking*, 10(1):12–26, 2002.
8. M. E. Gendy, A. Bose, S.-T. Park, and K. G. Shin. Paving the first mile for QoS-dependent applications and appliances. In *Proceedings of IWQoS*, 2004.
9. F. Hernandez-Campos, K. Jeffay, and F. D. Smith. Tracking the evolution of Web traffic: 1995-2003. In *Proceedings of MASCOTS*, 2003.
10. A. Kamra, V. Misra, and E. Nahum. Yaksha: A self tuning controller for managing the performance of 3-tiered websites. In *Proceedings of IWQoS*, 2004.
11. J. Kaur and H. Vin. Providing deterministic end-to-end fairness guarantees in core-stateless networks. In *Proceedings of IWQoS*, 2003.
12. B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, September 1999.
13. R. K. Mudi and N. R. Pal. A robust self-tuning scheme for PI- and PD-type fuzzy controllers. *IEEE Transactions on Fuzzy Systems*, 7(1):2–16, February 1999.
14. D. P. Olshefski, J. Nieh, and E. Nahum. ksniffer: Determining the remote client perceived response time from live packet streams. In *Proceedings of OSDI*, 2004.
15. K. Park, V. S. Pai, L. Peterson, and Z. Wang. CoDNS: Improving DNS performance and reliability via cooperative lookups. In *Proceedings of OSDI*, 2004.
16. S. Patchararungruang, S. K. halgamuge, and N. Shenoy. Optimized rule-based delay proportion adjustment for proportional differentiated services. *IEEE Journal on Selected Areas in Communications*, 23(2):261–276, February 2005.
17. V. Paxson and S. Floyd. Wide area traffic: The failure of possion modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.
18. L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of HotNets*, 2002.
19. L. Sha, X. Liu, Y. Lu, and T. F. Abdelzaher. Queueing model based network server performance control. In *Proceedings of RTSS*, 2002.
20. S. Shakkottai, R. Srikant, N. Brownlee, A. Broido, and K. Claffy. The RTT distribution of TCP flows in the Internet and its impact on TCP-based flow control. Technical report, The Cooperative Association for Internet Data Analysis (CAIDA), 2004.
21. F. G. Shinskey. *Process Control Systems: Application, Design, and Tuning*. McGraw-Hill, 4th edition, 1996.
22. J. Slottow, A. Shahriari, M. Stein, X. Chen, C. Thomas, and P. B. Ender. Instrumenting and tuning dataview—a networked application for navigating through large scientific datasets. *Software Practice and Experience*, 32(2):165–190, November 2002.
23. V. Sundaram and P. Shenoy. A practical learning-based approach for dynamic storage bandwidth allocation. In *Proceedings of IWQoS*, 2003.
24. J. Wei, X. Zhou, and C.-Z. Xu. Robust processing rate allocation for proportional slowdown differentiation on Internet servers. *IEEE Transactions on Computers*, 2005. In press.
25. M. Welsh and D. Culler. Adaptive overload control for busy Internet servers. In *Proceedings of USITS*, 2003.