*e*QoS: Provisioning of Client-Perceived 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 *client-perceived* end-to-end quality of service (QoS). Existing work, however, are limited to either server-side or network-side delays only. As a remedy, in this paper we propose a scalable and flexible framework, e*QoS*, to monitoring and controlling client-perceived QoS in web servers based on recently proposed approaches for real-time online QoS measurement. Within the *e*QoS, to deal with the inherent process delay in resource allocation and lack of accurate server model, we propose a model-independent two-level self-tuning fuzzy controller (STFC) to allocating server resources. To evaluate the performance of the *e*QoS framework, we have implemented a prototype in Linux. Through comprehensive experiments across wide-range server workload conditions using real-world and simulated networks, we demonstrate that provisioning of client-perceived QoS guarantees in heavy-loaded web servers is feasible and the *e*QoS is effective in such provisioning: The average deviation from a target response time 5 seconds with respect to whole web pages is only 1 second. We compare the STFC with non-adaptive fuzzy, linear proportional integral, and adaptive proportional integral controllers. The experimental results demonstrate that their deviations are around 125%, 175%, and 150% of that of the STFC, respectively.**

I. Introduction

The past decade has seen an increasing demand for provisioning of quality of service (QoS) guarantees to various network applications and clients. There exist many work on provisioning of QoS guarantees. Most of them, however, focus on web servers without considering network delays [1], [6], [7], [9], [41], [44], on individual network router [8], [15], [12], or on clients with assumptions of QoS supports in networks [14]. The focus of recent work is on end-to-end QoS guarantees in network cores [19], [39]. For example, in [19], the authors 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 not only affected by network delays but also by server delays. The objective of this paper is to guarantee *client-perceived* end-to-end QoS in web servers. To provide such QoS guarantees, service quality must accurately measured in real time so that server resources can be allocated promptly. The recent approach presented in [27] realizes such real-time measurement. It makes provisioning of client-perceived end-to-end QoS guarantees possible.

Our first contribution in this paper is that we propose *e*QoS, a 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 by taking advantage of the real-time QoS measurement. Moreover, in the framework, the guaranteed QoS is measured with respect to *whole* web pages instead of a single packet in networks or a single request [1], [2] or connection [23], [35] in web servers. It is because more than 50% of web pages have one or more embedded objects [16]. Our second contribution in the paper is that, within the *e*QoS, we propose a model-independent two-level self-tuning fuzzy controller (STFC) to allocate server resources.

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. 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 [1], [22], [33]. In these approaches, the nonlinear relationship between the allocated resource of a class and its received service quality is linearized at a fixed operating point. 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 their simple linearization thus is inappropriate.

The *process delay* is the latency between allocating server resources and accurately measured effect of the resource allocation on provided service quality. In web servers, resource allocation must be based on an accurately measured effect of previous resource allocation on the client-perceived response time of web pages. According to HTTP, to retrieve a web page, a client fist sends a request for the base page. The server then 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 thus is delayed. Consequently, the resource allocation is significantly complicated because it has to be based on an inaccurate measurement.

The process delay has also been addressed using queueing-model based predictor in [23], [35]. They integrated the predictor into a linear feedback controller to react to an incoming performance degradation according to predicted server workloads. Without an appropriate model to describe the server behaviors with respect to web pages, the performance of their

approach is limited.

The STFC is proposed to overcome the existing approaches' limitations. 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 server dynamics and unpredictability. On the second level is a scaling-factor controller. It aims to compensate the effect of process delay by adjusting the resource controller's output scaling factor according to transient server behaviors.

To evaluate the performance of the *e*QoS framework, we implement a prototype in Linux. We conduct experiments across wide-range server workload conditions on PlanetLab test bed [32]. We also evaluate the effect of network delays on its performance using simulated networks. The experimental results demonstrate that provisioning of client-perceived QoS guarantees is feasible and the *e*QoS is effective in such provisioning: The average deviation from a target response time 5 seconds with respect to whole web pages is only 1 second.

Within the *e*QoS framework, we also implement non-adaptive fuzzy, linear proportional integral (PI), and adaptive PI controllers and compare their performance with the STFC. The experimental results demonstrate that the STFC outperforms other controllers with much smaller deviations: the deviations of the non-adaptive fuzzy controller, the PI controller, and the adaptive PI controller are around 125%, 175%, and 150% of that of the STFC, respectively.

Our contributions in the paper can be summarized as follows.

1) Proposing a framework, *e*QoS, to guarantee client-perceived end-to-end QoS.
2) Proposing a model-independent two-level self-tuning fuzzy controller (STFC) for resource allocation in web servers.
3) Implementing and evaluating the performance of the *e*QoS on read-world and simulated networks and comparing the STFC with non-adaptive fuzzy, PI, and adaptive PI controllers.

The structure of the paper is as follows. Section II presents the structure of the *e*QoS framework and discusses the design and implementation issues. Section III presents the model-independent two-level STFC. Section IV evaluates the performance of the *e*QoS in real-world and simulated networks and compares the performance of different controllers. Section V reviews related work in provisioning of QoS guarantees in web servers and Section VI concludes the paper.

II. The *e*QoS Framework

The *e*QoS is designed to provide client-perceived end-to-end QoS guarantees in web servers. It monitors client-perceived QoS in real-time with respect to whole web pages with considerations of both network and server delays. To control client-perceived QoS, it dynamically allocates server resources between client classes by addressing the issues of lacking accurate server model and the process delay in resource allocation. We discuss the design and implementation issues in this section. Section III presents the design of the STFC.

A. The Design of the eQoS

The *e*QoS framework consists of four components: a web server, a QoS controller, a resource manager, and a QoS monitor, Figure 1 illustrates the components and their interactions. The Apache web server can be used to provide web services. The QoS controller aims to allocate resource allocation between client classes based on defined control rules. It can be any controller designed for the provisioning of QoS guarantees. For example, in our implementation, we realize the STFC, a non-adaptive fuzzy controller, a PI controller, and an adaptive PI controller.

The resource manager is to classify and manage client requests and to realize resource allocation between classes. In our design, it comprises of a classifier, several waiting queues, and a processing-rate allocator. The classifier determines 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) or be extended to use application-level information [42]. In an unmodified web server, a single waiting queue is created for a socket to store all established connections. In the *e*QoS, a request is stored in the accept queue corresponding to its client class within the resource manager. The requests from the same class are served in first-come-first-served manner. The process-rate allocator is to realize 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 [10], [44] can be easily integrated into the resource manager to ensure the server's aggregate performance.

The QoS monitor is to measure the response time in real-time with respect to whole web pages using similar ideas as presented in [27]. One challenge in measuring client-perceived response time with respect to web pages is how to dynamically determine the beginning and the ending of the web page during the transmission of client requests and server response. In the *e*QoS, the QoS monitor consists of a request-response collection component, a web reconstruction component, and a response-time measurement component. The request-response collection can capture live network packets or use information from instrumented web servers. Based on the collected information, the web pages are reconstructed in real time based on the HTTP *referer* field that specifies the web page from which the requested web object is obtained. The response time then is measured with considerations of network propagation delays and the effect of packet loss during the connection establishment [27].

The *e*QoS framework is scalable and flexible. Its scalability is achieved by employing a non-hierarchical or functionally symmetric architecture, which is inherently free of scaling bottlenecks. Therefore, the *e*QoS can be deployed in multiple servers independently. The flexibility comes from the inde-

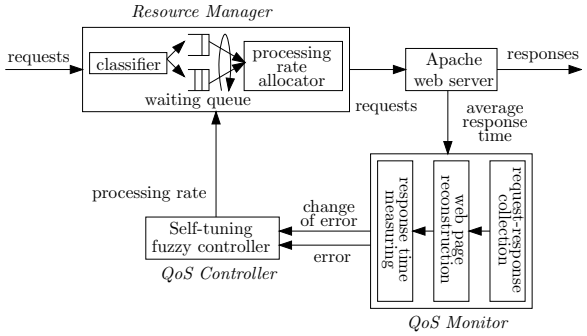


Fig. 1. The structure of the *e*QoS framework.

pendence between different components. For example, in the case of a web-server cluster, the resource manager can reside in every web servers while the QoS monitor and the QoS controller are deployed in a front-end proxy of the cluster to monitor and manage the resource of the whole cluster.

B. The Implementation of the e*QoS*

To evaluate the performance of the *e*QoS, we have implemented a prototype in Linux. In our implementation, we take into account following four issues. First, in a heavily loaded web server, a new connection request may be dropped by the operating system due to overflowed accept queue in the kernel and the exponential back-off mechanism of client-side TCP will be triggered. To minimize such possibility so as to reduce the retransmission delays, our implementation of the resource manager drains the kernel's accept queue in a tight loop. The accepted connections are stored in their corresponding accept queues that have unlimited size within the resource manager.

Second, to ensure that the aggregate performance of a web server to be unaffected by supporting QoS guarantees, the processing-rate allocator realizes the resource allocation using work-conserving weighted fair queueing algorithms [28]. In the implementation, the Apache web server is modified to accept requests from the resource manager through a unix domain socket. When a child process in the Apache web server calls *accept()* on the unix domain socket, a signal is sent to the processing-rate allocator. Upon receiving the signal, the allocator determines which class should be served and dispatches a request from the class through the unix domain socket to the child process. In the case that all accept queues are empty, a flag is set to indicate that there exists an idle child process. A newly arrived request will be passed to the child process immediately if the flag is set.

In addition, similar as the approaches in [44], in order to prevent sudden spikes in the response time sample from causing oscillations in the resource allocation, the average response time is smoothed using an exponential weighted moving average with parameter β:

$$W(k) = \beta \cdot W(k) + (1 - \beta) \cdot W(k-1),$$

where $W(k)$ is the average response time computed in sampling period k. We have carried experiments with different β and found no qualitative differences. To balance weights between current served requests and those processed in the past, in the implementation we set β to 0.5.

Finally, in the *e*QoS, we assume that the capacity of a web server is limited by its CPU (processing rate). With the popularity of dynamic web content, such as those in e-Commerce web servers, processing rate becomes easier to be bottleneck resource than others. Similar assumptions have also been adopted in previous work [2], [22]. Note that although we assume that processing rate is the bottleneck of a web server, the main ideas can be applied for alternative bottleneck resources as well.

III. The Self-Tuning Fuzzy Controller

To guarantee client-perceived QoS effectively, the QoS controller must address issues of the process delay in resource allocation and lack of accurate server model. In the section, we first briefly review the interactions between clients and web servers during the retrieval of web pages, followed by the process delay and its effect on resource allocation. After that, we present our design of the STFC in details.

A. The Interactions between Clients and Web Servers

Client-perceived response time of a web page 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. Figure 2 shows the interactions between clients and web servers during the retrieval of *http://www.foo.com/index.html.*

1) The client first obtains the IP address of *www.foo.com* by inquiring domain name servers or from its own cache if the web site has been accessed in the past and the address is cached.
2) The client sends a connection request to the web server corresponding to the IP address and establishes a TCP connection via three-way handshake.
3) After establishing the TCP connection, the client sends an HTTP request for *index.html.* Note that in practice, the client normally sends the first HTTP request immediately (within 0.5 *ms* in our experiments) after the last step of the three-way handshake.
4) The server determines when the established TCP connection should be passed to the Apache web server based on its resource allocation.
5) A child process of the Apache web server processes the request and sends *index.html* back to the client.
6) The client sends individual request for each embedded object to the server.
7) The server sends all embedded objects back to the client.
8) The server waits for possible requests from the same connection for certain period (the default is 15 seconds in the Apache web server) and then terminates the connection.

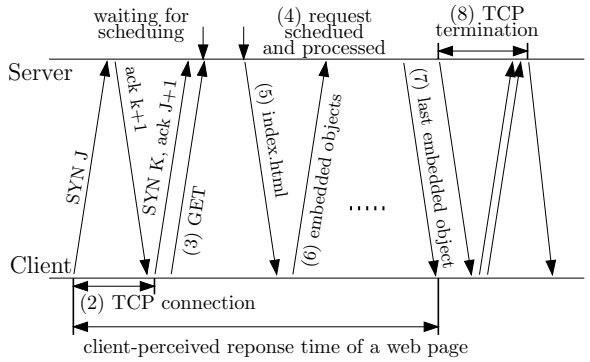


Fig. 2. The interactions between the client and the web server during the retrieval of *http://www.foo.com/index.html*.

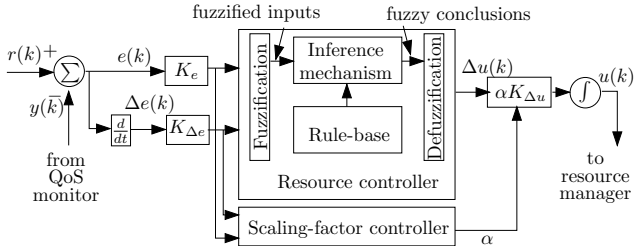


Fig. 4. The structure of the STFC.

In this work we only consider the latency incurred in step (2) through step (7). The latency incurred in step (1) is only measurable from client side and is difficult to be obtained from web servers. More importantly, if the server address is available in the client side (the client's cache or client-side domain name servers), the latency is normally negligible in comparison with response time of web pages, which is normally in the order of seconds. For example, as shown in [29], 70% of the name-lookup requests have response time less than 10 *ms* and 90% of them are less than 100 *ms* for all of their examined domain name servers except one.

In the *e*QoS, we aim to guarantee the average response time of web pages perceived by premium clients to be close to a pre-defined reference value $D(k)$. We have

$$W(k) = D(k). \tag{1}$$

By achieving this, the *e*QoS guarantees QoS of premium clients and provides as good as possible services to other clients simultaneously. Because the server load can grow arbitrary high, it is impossible to guarantee QoS of all clients under heavy-load conditions.

B. The Process Delay in Resource Allocation

To provide QoS guarantees, the resource allocation in web servers must 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 in step (4). Aforementioned, there exists process delay between the resource allocation and the effect measurement. It has been recognized as one of the most difficult dynamic element naturally occurring in physical systems to deal with [37]. 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. Consequently, the resource allocation must be designed to compensate the process-delay effect.

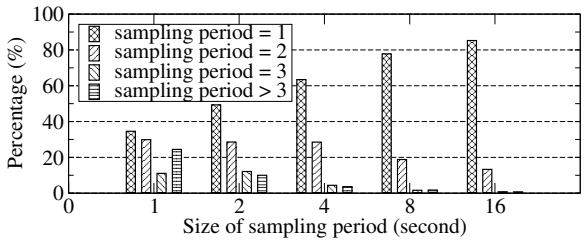
We have conducted experiments to quantify the process delay in resource allocation. The experimental environments are described in Section IV-A. In the experiments, the number of concurrent users and the RTT are set to 700 and 180 *ms*, respectively. Figure 3(a) shows the percentage of requests finished within different numbers of sampling period after being admitted. Figure 3(b) depicts corresponding cumulative distribution function of the service time, which is the latency incurred in step (5) through (7). Because the latency incurred in step (2) through (4) can be measured at the end of step (4), it does not affect the accuracy of measured resource-allocation effect. Comparing Figure 3(a) and Figure 3(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 several sampling periods ago. It further complicates the accuracy of effect measurement. Consequently, the resource-allocation effect cannot be accurately measured promptly.

C. The Structure of the Resource Controller

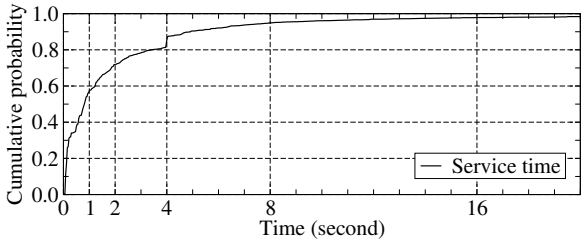
Within the *e*QoS framework, we propose a model-independent two-level STFC to controlling the resource allocation in web servers. Figure 4 presents the structure of the STFC. The resource controller on the first level takes advantage of fuzzy control theory to address the issue of lacking accurate server models. The scaling-factor controller is to compensate the effect of process delay by adjusting the resource controller's output scaling factor according to transient server behaviors.

In the resource controller, the resource allocated to premium class in sampling period $k+1$, denoted by $u(k+1)$, is adjusted according to its error $e(k)$ (i.e., the difference between the reference value and the achieved one) and change of error $\Delta e(k)$ in previous sampling period k using a set of control rules about heuristic control knowledge. In the controller, $e(k)$ and $\Delta e(k)$ are calculated using the reference value $r(k)$ and the achieved value $y(k)$. Based on these, the controller calculates resource adjustment $\Delta u(k)$ for next sampling period, which is then fed into the resource manager component.

As shown in Figure 4, 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



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



(b) The cumulative distribution function of service time with respect to web pages.

Fig. 3. The process delay in resource allocation.

the change of resource of premium class in numeric value.

The resource controller presented in Figure 4 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 is

$$u(k+1) = u(k) + \alpha K_{\Delta u}\Delta u(k) = \int \alpha K_{\Delta u}\Delta u(k)dk. \quad (2)$$

Note that these scaling factors are positive in order to ensure the stability of the control system, which is proved in [43].

The parameters of the control loop as shown in Figure 4 are defined as follows.

$$r(k) = D(k), \quad (3)$$
$$y(k) = W(k), \quad (4)$$
$$e(k) = D(k) - W(k), \quad (5)$$
$$\Delta e(k) = e(k) - e(k-1). \quad (6)$$

1) Design of the Rule-base: 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 how to translate such heuristic control knowledge into a set of control rules so that it is able 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 Figure 5(a). Note that they indicate the sign and the size in relation to the other linguistic values. This gives more flexibility to the STFC than other control-theoretic approaches.

We next analyze the effect of the controller on the provided services as shown in Figure 6(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,

Zone 1: $e(k)$ is positive and $\Delta e(k$ is negative;
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,

Zone 2: $e(k)$ is negative and $\Delta e(k)$ is negative;
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.

By identifying these five zones, we are able to design the fuzzy control rules. Let $U(k)$ denote the equilibrium resource value of premium class at which the reference value can be achieved. Let $\tilde{u}(k)$ denote the difference between the equilibrium resource value and current one. It follows that

$$\tilde{u}(k) = U(k) - u(k). \quad (7)$$

In order to provide QoS guarantees, the resource allocation should converge to its equilibrium value. Based on the characteristics of the five zones shown in Figure 6(a), in linguistics, we have

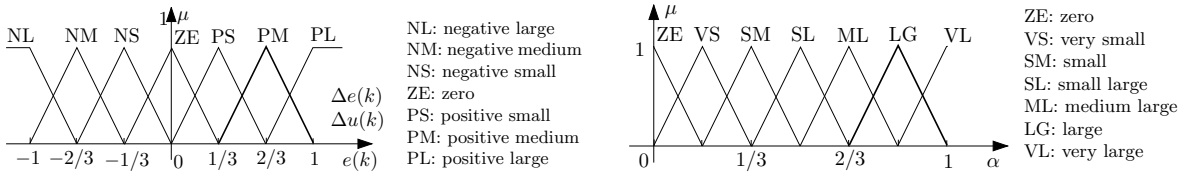
$$\text{“}\tilde{u}(k)\text{”} = -[\text{“}e(k)\text{”} + \text{“}\Delta e(k)\text{”}], \quad (8)$$

and "$\Delta u(k)$" is set as

$$\text{“}\Delta u(k)\text{”} = \text{“}\tilde{u}(k)\text{”}. \quad (9)$$

Notice that we are using linguistic values of the controller's inputs and output, and NL and PL are their lower bound and upper bound, respectively. Therefore, when "$e(k)$" and "$\Delta e(k)$" are NL, "$\Delta u(k)$" is set to PL.

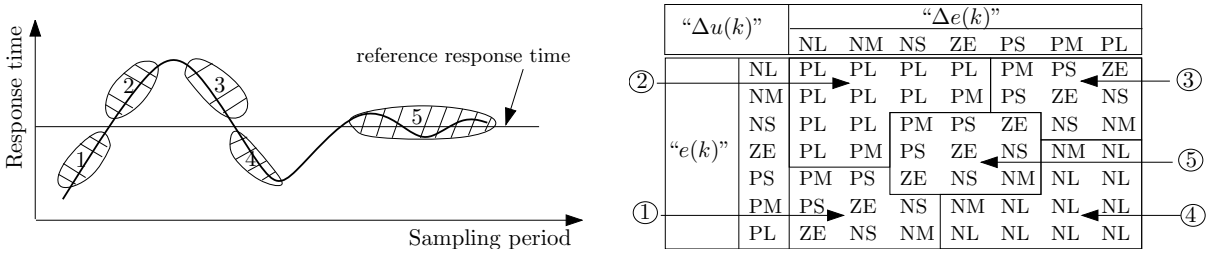
The resulted control rules are summarized in Figure 6(b). A general linguistic form of these rules is read as: *If* premise *Then* consequent. Let $rule(m,n)$, where m and n assume linguistic values, denote the rule of the (m,n) position in



(a) The membership functions of e, Δe, and Δu.

(b) The membership functions of α.

Fig. 5. The membership functions of the STFC.



(a) Illustration of control effect.

"$\Delta u(k)$"	"$e(k)$"	NL	NM	NS	ZE	PS	PM	PL
	NL	PL	PL	PL	PL	PM	PS	ZE
	NM	PL	PL	PL	PM	PS	ZE	NS
	NS	PL	PL	PM	PS	ZE	NS	NM
	ZE	PL	PM	PS	ZE	NS	NM	NL
	PS	PM	PS	ZE	NS	NM	NL	NL
	PM	PS	ZE	NS	NM	NL	NL	NL
	PL	ZE	NS	NM	NL	NL	NL	NL

(b) The rule-base of the resource controller.

Fig. 6. Fuzzy control rules in the resource controller.

Figure 6(b). As an example, $rule(PS, PS) = NM$ reads that: *If* the error is positive small *and* the change of error is positive small *Then* the change of resource is negative medium.

Remark: 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 model for web servers. Therefore, the STFC is *model independent*.

2) Fuzzy Quantification of the Rule-base: 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 Figure 5(a). We have also examined membership functions with different shapes, such as "Gaussian" and "trapezoid", and found no significant performance differences between them. For brevity, in the paper we only present the results due to the "triangle" membership functions. In Figure 5(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. Let $\mu_m(e(k))$ denote the certainty of $e(k)$ of the mth membership function, and $\mu_n(\Delta e(k))$ the certainty of $\Delta e(k)$ of the nth membership function. When $e(k) = 1/12$ and $\Delta e(k) = 1/3$, according to the membership functions of $e(k)$ and $\Delta e(k)$, all membership functions yield 0 except that $\mu_{ZE}(e(k)) = 0.75$, $\mu_{PS}(e(k)) = 0.25$, and $\mu_{PS}(\Delta e(k)) = 1$.

3) Inference Mechanism and Defuzzification: The inference mechanism is to determine which rules should be activated and what conclusions can be reached. Let $\mu(m, n)$ denote the premise certainty of $rule(m, n)$. The *and* operation in the premise is calculated via *minimum*. Suppose $e(k) = 1/12$ and $\Delta e(k) = 1/3$, then only two rules, $rule(ZE, PS)$ and $rule(PS, PS)$, should be activated because the premise certainties of all other rules are 0 according to the example presented in Section III-C.2. Consequently,

$\mu(ZE, PS) = \min\{0.75, 1\} = 0.75$, and $\mu(PS, PS) = \min\{0.25, 1\} = 0.25$.

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. Let $b(m, n)$ denote the center of membership function of the consequent of $rule(m, n)$. In this case, it is where the membership function reaches its peak. The fuzzy control output is

$$\Delta u(k) = \frac{\sum_{m,n} b(m, n) \cdot \mu(m, n)}{\sum_{m,n} \mu(m, n)}. \tag{10}$$

D. The Scaling-factor Controller

To successfully design the resource controller discussed in Section III-C, a proper output scaling factor $\alpha K_{\Delta u}$ is important because of its global effect on the control performance. Furthermore, the design of the STFC needs to take into account the process delay as discussed in Section III-B. 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 [25] so as to compensate the effect of the process delay. Notice that it is difficult to apply traditional methods, such as gradient descent method, to adjust $\alpha K_{\Delta u}$ without an accurate server model.

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 Figure 5(b). Because α needs to be positive to ensure the stability of the control system, "α" assumes different linguistic values from "$e(k)$" and "$\Delta e(k)$". Figure 5(b) also shows the linguistic values and their meanings.

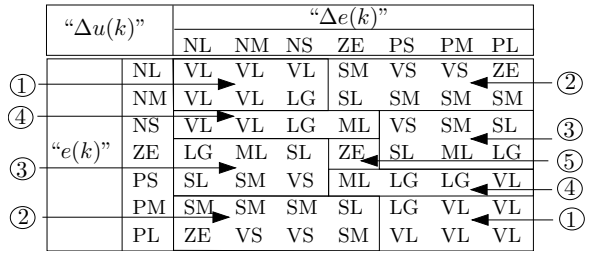


Fig. 7. The rule-base of the scaling-factor controller.

The control rules of the scaling-factor controller are designed in order to compensate the effect of the process delay on the performance of the STFC. They are shown in Figure 7 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, it means the average response time of premium class just reaches the reference value and is moving away upward. 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. Such large variation also justifies the need for dynamic output scaling factor.
4) Note that the workload of a web server is highly dynamic and has disturbances. The scaling-factor controller also provides regulation against the disturbances. For example, 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 need to tune the K_e, $K_{\Delta e}$, and $K_{\Delta u}$ through trials and errors. In the step, the scaling-factor controller is turned off and α is set to 1. Notice that the tuning of control factors is required for all practical controllers because no fixed values fit all situations [13]. 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. In addition, as suggested in [25], 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. The K_e and $K_{\Delta e}$ are kept unchanged.

Finally we remark that the STFC has small overhead. For any inputs, only two membership functions lead to nonzero values in the resource controller. Therefore, at most four rules are on at any time in the resource controller. Similarly, at most four rules are on in the scaling-factor controller. It is demonstrated in Section IV-F that there is negligible performance difference with the STFC on and off. Furthermore, the implementation complexity is small: our implementation of the STFC totaled less than 100 lines of C code.

IV. Performance Evaluations

We define relative deviation $R(e)$, which is based on root-mean square error that is one of the most widely utilized performance criteria [24], as the metric to measure the performance of the *e*QoS. We have

$$R(e) = \frac{\sqrt{\sum_{k=1}^{n} \left(D(k) - W(k)\right)^2 / n}}{D(k)} = \frac{\sqrt{\sum_{k=1}^{n} e(k)^2 / n}}{D(k)}. \tag{11}$$

The relative deviation describes the size of deviation from the reference value $D(k)$. Furthermore, because large deviation contributes heavily to $R(e)$, it reflects the transient characteristics of a control system. Thus, the smaller the $R(e)$, the more the achieved average response time concentrates near the reference value and the better the controller's performance.

A. Experimental Environments and Configurations

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 United Kingdom).

The server workload was generated by SURGE [3]. It was controlled by adjusting the number of concurrent user equivalents (UEs) in SURGE. Notice that the fixed number of UEs does not affect the representativeness (i.e., self-similarity and high dynamics) of the generated web traffic [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 SURGE was set up without pipelining. It is because in practice most web browsers, including Microsoft Internet Explorer 6.0, serialize requests so that the next one is sent only after receiving preceding request's response.

The Apache web server was used to provide web services. It was set up with support of HTTP/1.1. The number of the maximal concurrent child processes was set to 128. Since QoS guarantees is most necessary when the server is heavily loaded, we set up the environment such that the ratio between the number of UEs and the number of child processes could drive the server to be heavily loaded. Notice that although large web servers such as e-Commerce servers usually have more child processes than we configured, they also tend to have much more clients than we simulated. Therefore, our configuration can be viewed as an emulation of real-world heavy-load scenarios at a small scale [22].

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 reference values of class 1 and class 2 were 5 and 11 seconds, respectively. As pointed out in [5], the service quality of web servers is rated as "good" and "average" when the average response time of web pages is less than 5 seconds and 11 seconds, respectively. Notice that the number of classes in real environments is usually limited. As recommended in [26], a service provider needs to support 2 or 3 different levels of services.

We aimed to provide guaranteed service *only* when the server is heavily loaded. In our experiment configurations, 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. Thus, it is meaningful only when the number of UEs is no less than 500. When the number of UEs is larger than 800, we have observed refused connections using unmodified Apache web server. In such case, the server becomes overloaded and admission control mechanisms, such as those presented in [10], [41], [44], should be used to ensure the aggregate performance of the web server.

To investigate the effect of network latency on the performance of the *e*QoS, we have implemented a network-delay simulator. It is to emulate wide-area network delay in a similar way to [38] and dummynet [34]. In the simulated environment, two machines are used as clients and one as the network simulator. They have the same hardware configurations as the server and are connected by a 100 Mbps Ethernet. We changed the network routing in the server and client machines so that the packets between them were sent to the simulator. Upon receiving a packet, the simulator routes the packet to an "ethertap" device. A small user-space program reads the packet from the "ethertap" device, delays it, and writes it back to the device. The packet is then routed to the ethernet. Thus, we can control the RTT between the server and clients. Our evaluation experiments show that the simulator is effective in emulating wide-area network delay. For example, with the RTT set as 180 *ms*, ping times were showing a round trip of around 182 *ms*.

In the experiments on the simulated networks, the RTT between clients and servers was set to be 40, 80, or 180 *ms*. They represent the transmission 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 [36].

B. *Effectiveness of the* e*QoS*

To evaluate the effectiveness of the *e*QoS in providing client-perceived QoS guarantees, 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. After that, the controller was turned on. The size of sampling period is set to 4 seconds. The effect of the sampling period on the performance of the *e*QoS is discussed in Section IV-E. Figure 8 presents the relative deviation of provided average web-page response times in the experiments.

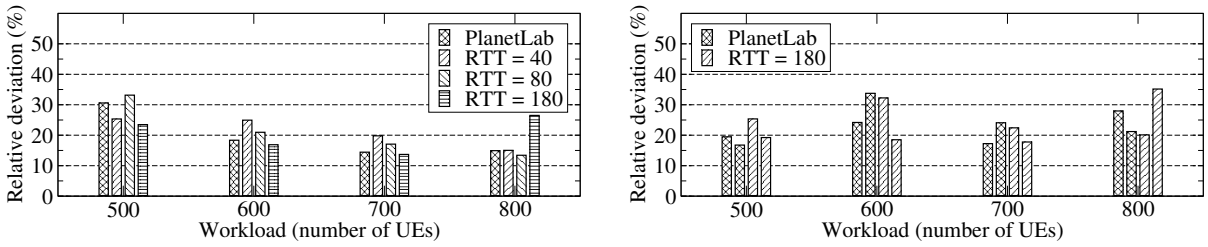
Figure 8(a) shows the relative deviations of the premium class relative to the reference value (5 seconds). From the figure we observe that all the relative deviations are smaller than 35%. Meanwhile, most of them are around 20%. It means the size of deviations is normally around 1.0 seconds.

Figure 8(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%. Because the reference values of class 1 and class 2 are 5.0 and 11.0 seconds, the results indicate the deviation sizes are between 0.75 and 1.5 seconds for class 1 and between 1.65 and 3.3 seconds for class 2. Note that the results shown in Figure 8 are due to experiments conducted under different network conditions and server workloads. We conclude from the results that the *e*QoS is able to guarantee client-perceived QoS effectively in various environments.

C. *Feasibility of Client-Perceived QoS Guarantees*

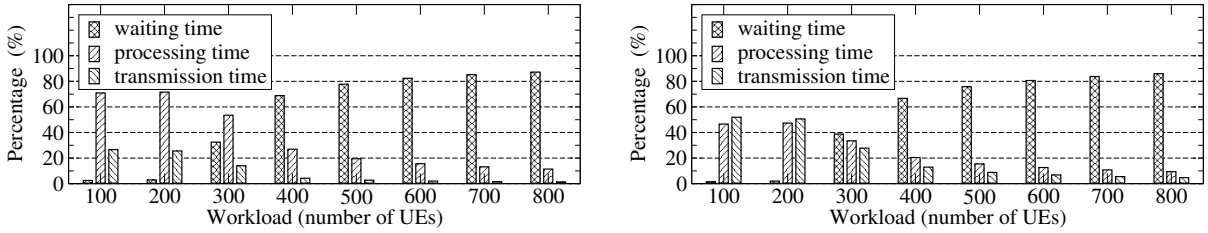
In this subsection we investigate why it is feasible to guarantee client-perceived QoS from server side under heavy-load conditions. The client-perceived response time consists of waiting time, processing time, and transmission time. The waiting time is the latency incurred in step (4) as shown in Section III-A. It is the time interval that starts when a connection is accepted by the server 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 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. We conducted experiments with different workloads and network delays. Figure 9 shows the breakdown of client-perceived response time under different network latency. For brevity, we omit the results where RTT was set to 80 *ms*.

From Figure 9 we observe that, when the server is heavily loaded (the number of UEs is larger than 400), the server-side waiting time is the dominant part of client-perceived response time. The finding is consistent with others, such as those in [4], [22], [33]. It is because that, when the server is



(a) Two classes. (b) Three classes.

Fig. 8. The performance of the *e*QoS with two and three classes.



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

Fig. 9. The breakdown of response time of web pages under different RTTs.

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 are still able to control the client-perceived response time by controlling the server-side waiting time and processing time. The server-side QoS guarantees in heavily loaded web servers thus is feasible in practice.

As observed in [11], network utilization is normally low. In the work, thus, we assume the network is not highly congested. Otherwise the RTT may become too large to provide QoS guarantees even with lightly loaded web servers. For example, assuming RTT is 1000 *ms*, the retrieval of a web page with 10 embedded objects via one TCP connection takes at least 12 seconds: 1.5 seconds for establishing TCP connection and 1 second for retrieving the base HTML file and every embedded object with serialized requests.

D. Comparison with Other Controllers

Within the *e*QoS framework, we also implement three controllers: a fuzzy controller without self-tuning, a traditional PI controller, and an adaptive PI controller using the basic idea of [17]. The selection of linear PI controllers is because two reasons: the PI controllers are widely used, including [17], [22]; the STFC is a PI-like controller with nonlinear operating functions so that the comparison is fair. In [17], the parameters of the adaptive PI controller are adjusted according to the admission probability of a class. In our implementation, the parameters are adjusted according to the processing rate of a class in a similar manner.

We have specifically tuned the fuzzy controller in an environment that the number of UEs was set to 700 and RTT was assumed to be 180 *ms* on the simulated networks. We also tuned the PI controller in the same environment under the guideline of Ziegler-Nichols method [13]. After that, we then conducted experiments with different server workloads and network conditions on PlanetLab and simulated networks.

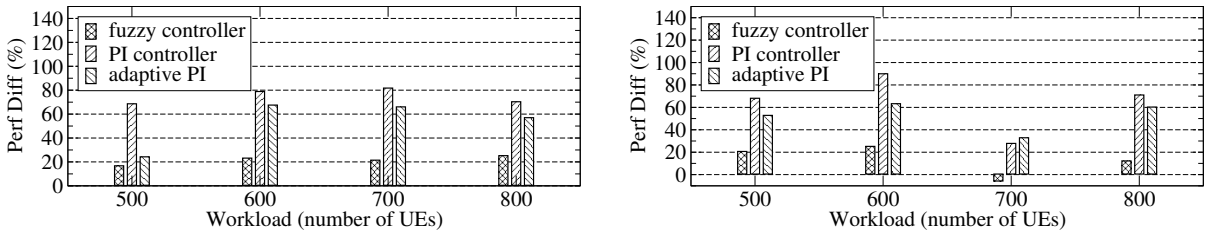
Taking the performance of STFC as a baseline, we define the performance difference between the STFC and other controller as

$$PerDiff = \frac{R(e)_{other} - R(e)_{STFC}}{R(e)_{STFC}}, \tag{12}$$

where $R(e)_{other}$ and $R(e)_{STFC}$ are the relative deviations of other controller and the STFC, respectively. If $PerDiff$ is positive, the STFC has better performance than other controller and vice versa. Figure 10 presents the performance difference of the fuzzy controller, the PI controller, and the adaptive PI controller. Due to space limitation, we only present the results where RTT was set to 180 *ms*.

From Figure 10(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 some environment. Even under such environment, the performance difference between the STFC and the fuzzy controller is negligible; that is, the performance difference is just -6%.

Under all other conditions, the STFC provides better services than the non-adaptive fuzzy controller. That is, the average performance difference is about 25%. Such behavior can be observed from Figure 10(b) and Figure 10(a). Aforementioned, the STFC further adjusts the output scaling factor of the fuzzy controller adaptively according to the transient behaviors of the web server. Such tuning is important to



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

Fig. 10. The performance comparison in PlanetLab and simulated networks.

compensate the effect of the process delay in resource allocation. Thus, the STFC is able to provide services with smaller relative deviation than the fuzzy controller. The observation also demonstrates that our analysis and design of the self-tuning scaling-factor controller are correct.

In comparison with the PI controller, the STFC achieves better performance even when the PI controller operates under its specifically tuned environment. It can be observed in Figure 10(b). When the number of UEs is 700 and RTT is 180 *ms*, their performance difference is 28%. From Figure 10 we observe that all performance differences of the PI controller are larger than 60% and the average is around 75%. The poor performance of the PI controller is due to its inaccurate underlying model. In the PI controller, we follow the approach in [17] 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 [31]. 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%. Moreover, the PI and adaptive PI controllers provide no means to compensate the effect of the process delay in resource allocation.

E. Effect of the Sampling Period on QoS Guarantees

In the subsection we investigate the effect of the sampling period on the performance of the *e*QoS and determine an appropriate size. We carried out experiments with different settings of the sampling period on both PlanetLab and simulated networks. For brevity we only show the results from PlanetLab in Figure 11.

From the figure we observe that the relative deviation decreases with the increase of the sampling-period size. As shown in Figure 3(a), the percentage of requests finished within the admitted sampling periods increases with the increase of sampling-period size. Therefore, the controller is able to measure the resource-allocation effect on client-perceived response time more accurately with the increase of the sampling period.

When the sampling-period size continues to increase, the relative deviation increases. It is because that, with a large sampling period, the processing rate of premium class is adjusted less frequently than a small one. Consequently, the

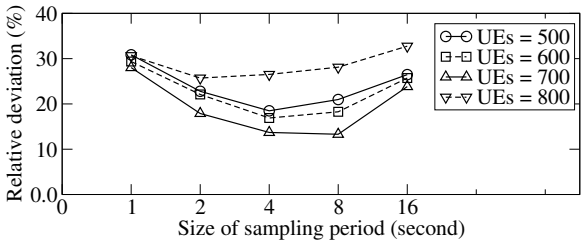


Fig. 11. The effect of sampling period on the performance of the *e*QoS.

*e*QoS becomes less adaptive to the transient workload disturbances and the relative deviation increases. Based on the results shown in Figure 11, in our experiments, we set the size of sampling period to 4 seconds. Notice that we do not claim our setting of sampling period is optimal. How to determine an optimal sampling period is part of our future work.

F. Effect of the e*QoS on Server Performance*

In this subsection, we investigate the effect of the *e*QoS on the performance of web servers. We conducted experiments with the *e*QoS (STFC on and STFC off). We conducted experiments with different numbers of UEs for 10 minutes on PlanetLab. To obtain an accurate average response time, the reported results are the average without the beginning and the ending periods: that is, the response time is averaged from the 2nd to the 9th minute. Due to space limitation, we summerize our observations. We observe that the performance differences between the STFC (off) and the STFC (on) is within 1%. It indicates that the overhead of the STFC itself is small because the controller only needs to adjust resource allocation once a sampling period. It also indicates that the aggregate server performance is not affected by the working-conserving processing-rate allocation.

V. Related Work

Provisioning of QoS guarantees has been an active research topic. In practice, the client-perceived service quality is mainly determined by both networks and web servers. Moreover, the service quality is normally measured with respect to whole web pages. The existing approaches, however, measured service quality with respect to a single packet in networks or an individual request [1], [2], [6], [9], [17], [33] or connection [22], [23], [35], [41] in web servers. In comparison, the

proposed framework *e*QoS aims to guarantee the QoS from the perspective of end clients.

Early work focused on providing differentiated services to different client classes using priority-based scheduling. For example, in [2], the authors aimed to provide better services to premium class than basic class by adjusting the number or the priority of the allocated processes between the classes on either user level or on kernel level. They, however, were unable to guarantee the QoS a class received.

To guarantee the QoS of a class, queueing-theoretic approaches have been proposed. It is well known that the delay upper bound in a $G/G/1$ is determined by the system load and the variance of requests' inter-arrival and service time distributions. In the approach presented in [41], the load of a class is adjusted by controlling its resource allocation so that the target delay equals to the upper bound. Its performance highly depends on the parameter estimation, such as the variance, which is difficult to be accurate.

Traditional linear feedback control has also been applied to control the resource allocation in web servers [1], [22], [33]. Because the behavior of a web server changes continuously,, the performance of the linear feedback control is limited. In comparison, our approach is model independent by taking advantage of fuzzy control theory to manage the server-resource allocation based on the analysis of resource-allocation on achieved response time.

Recent work have applied adaptive control [17], [18] and machine-learning [40] to address the lack of accurate server model. For example, in [18], an adaptive controller based on dynamically estimated system model is proposed. 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. For example, in [20], the authors presented a fuzzy control model to address the non-linear QoS requirements of different multimedia applications under different resource constraints. Similarly, in [21], because of the nonlinearity of web servers, fuzzy control theory is also used to determine an optimal number of concurrent child processes to improve the aggregated server performance. In [30], it is applied to dynamically adjust the delay ratios between different traffic flows in proportional delay differentiation service model to compensate the effect of the traffic bursty on the delay of premium class. The objective of the STFC is different in that its focus is on providing client-perceived end-to-end QoS guarantees. Moreover, the STFC explicitly addresses the inherent process delay in resource allocation.

VI. Conclusions

In the paper, we have proposed framework *e*QoS, which to the best of our knowledge is the first one to providing *client-perceived* end-to-end QoS guarantees, based on real-time response time measurement. Within the framework, we have proposed a model-independent two-level STFC that explicitly addresses the process delay in resource allocation. To evaluate the performance of the *e*QoS, we have implemented the framework in Linux and carried out comprehensive experiments under different workload conditions using real-world networks on PlanetLab test bed and simulated networks. The experimental results have shown that it is feasible to provide client-perceived QoS guarantees in heavy-load web servers. They also demonstrated the effectiveness of the *e*QoS and the superiority of the STFC over non-adaptive fuzzy controller, traditional PI controller, and adaptive PI controller with much smaller deviations.

In this work, we focus on single-tiered web servers. Because of the popularity of multi-tiered e-Commerce web sites, in our future work we will investigate how to incorporate the *e*QoS into a multi-tiered environments and its performance on dynamic content.

References

[1] Tarek F. Abdelzaher, Kang G. Shin, and Nina 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] Jussara Almeida, Mihaela Dabu, Anand Manikutty, and Pei Cao. Providing differentiated levels of service in web content hosting. In *Proceedings of ACM SIGMETRICS Workshop on Internet Server Performance*, pages 91–102, June 1998.

[3] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM Sigmetrics '98 Conference*, pages 151–160, June 1998.

[4] Paul Barford and Mark Crovella. Critical path analysis of TCP transactions. *IEEE/ACM Transactions on Networking*, 9(3):238–248, 2001.

[5] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into Web server design. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 1–16, 2000.

[6] Nina Bhatti and Rich Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, 1999.

[7] Preeti Bhoj, Srinivas Ramanathan, and Sharad Singhal. Web2K: Bringing QoS to web servers. Technical Report HPL-2000-61, HP Laboratories, May 2000.

[8] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. *An Architecture for Differentiated Services*. IETF, Request for Comments 2475, December 1998.

[9] Josep M. Blanquer, Antoni Batchelli, Klaus Schauser, and Rich Wolski. Quorum: Flexible quality of service for Internet services. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[10] Xianping Chen and Prasant Mohapatra. Performance evaluation of service differentiating Internet servers. *IEEE Transactions on Computers*, 51(11):1368–1375, November 2002.

[11] Baek-Young Choi, Sue Moon, Zhi-Li Zhang, Konstantina Papagiannaki, and Christophe Diot. Analysis of point-to-point packet delay in an operational network. In *Proceedings of IEEE Infocom*, Hong Kong, March 2004.

[12] Constantinos Dovrolis, Dimitrios Stiliadis, and Parameswaran Ramanathan. Proportional differentiated services: Delay differentiation and packet scheduling. *IEEE/ACM Transactions on Networking*, 10(1):12–26, 2002.

[13] Gene F. Franklin, J. David Powell, and Abbas Emami-naeini. *Feedback Control of Dynamic Systems*. Prentice Hall, 4th edition, 2002.

[14] Mohamed El Gendy, Abhijit Bose, Seong-Taek Park, and Kang G. Shin. Paving the first mile for QoS-dependent applications and appliances. In *Proceedings of International Workshop on Quality of Service (IWQoS)*, pages 245–254, 2004.

[15] Dan Grossman. *New Terminology and Clarifications for Diffserv*. Network Working Group, Request for Comments 3260, April 2002.

[16] Felix Hernandez-Campos, Kevin Jeffay, and F. Donelson Smith. Tracking the evolution of web traffic: 1995-2003. In *Proceedings of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 16–25, 2003.

[17] Abhinav Kamra, Vishal Misra, and Erich Nahum. Yaksha: A self tuning controller for managing the performance of 3-tiered websites. In *Proceedings of International Workshop on Quality of Service (IWQoS)*, pages 47–56, 2004.

[18] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proceedings of International Workshop on Quality of Service (IWQoS)*, pages 67–76, 2004.

[19] Jasleen Kaur and Harrick Vin. Providing deterministic end-to-end fairness guarantees in core-stateless networks. In *Proceedings of International Workshop on Quality of Service (IWQoS)*, pages 401–421, 2003.

[20] Baochun Li and Klara Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, September 1999.

[21] Xue Liu, Lui Sha, Yixin Diao, Joesph L. Hellerstein, and Sujay Parekh. Online response time optimization of apache web server. In *Proceedings of International Workshop on Quality of Service (IWQoS)*, pages 461–478, 2003.

[22] Chenyang Lu, Tarek F. Abdelzaher, John A. Sankovic, and Sang H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2001.

[23] Ying Lu, Tarek F. Abdelzaher, Chenyang Lu, Lui Sha, and Xue Liu. Feedback control with queueing-theoretic prediction for relative delay guarantees in web servers. In *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 208–217, May 2003.

[24] Dimitris G. Manolakis, Vinay K. Ingle, and Steplen M. Kogon. *Statistical and Adaptive Signal Processing*. The McGraw-Hill Companies, 2000.

[25] Rajani K. Mudi and Nikhil 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.

[26] Kathleen Nichols, Van Jacobson, and Lixia Zhang. *A Two-bit Differentiated Services Architecture for the Internet*. IETF, Request for Comments 2638, July 1999.

[27] David P. Olshefski, Jason Nieh, and Erich Nahum. ksniffer: Determining the remote client perceived response time from live packet streams. In *Proceedings of Usenix Operating Systems Design and Implementation (OSDI)*, 2004.

[28] Abhay K. Parekh and Robert G. Gallager. 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, 1993.

[29] KyoungSoo Park, Vivek S. Pai, Larry Peterson, and Zhe Wang. CoDNS: Improving DNS performance and reliability via cooperative lookups. In *Proceedings of Usenix Operating Systems Design and Implementation (OSDI)*, 2004.

[30] Sunthiti Patchararungruang, Saman K. halgamuge, and Nirmala Shenoy. Optimized rule-based delay proportion adjustment for proportional differentiated services. *IEEE Journal on Selected Areas in Communications*, 23(2):261–276, February 2005.

[31] Vern Paxson and Sally Floyd. Wide area traffic: The failure of possion modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[32] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of ACM Workshop on Hot Topics in Networking (HotNets)*, 2002.

[33] Prashant Pradhan, Renu Tewari, Sambit Sahu, Abhishek Chandra, and Prashant Shenoy. An observation-based approach towards self-managing web servers. In *Proceedings of International Workshop on Quality of Service (IWQoS)*, 2002.

[34] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *SIGCOMM Computer Communnication Review*, 27(1):31–41, 1997.

[35] Lui Sha, Xue Liu, Ying Lu, and Tarek F. Abdelzaher. Queueing model based network server performance control. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, pages 81–90, 2002.

[36] Srinivas Shakkottai, R. Srikant, Nevil Brownlee, Andre 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.

[37] Francis Greg Shinskey. *Process Control Systems: Application, Design, and Tuning*. McGraw-Hill, 4th edition, 1996.

[38] Joan Slottow, Ali Shahriari, Michael Stein, Xiao Chen, Chris Thomas, and Philip 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.

[39] Wei Sun and Kang G. Shin. Coordinated aggregate scheduling for improving end-to-end delay performance. In *Proceedings of International Workshop on Quality of Service (IWQoS)*, 2004.

[40] Vijay Sundaram and Prashant Shenoy. A practical learning-based approach for dynamic storage bandwidth allocation. In *Proceedings of International Workshop on Quality of Service (IWQoS)*, 2003.

[41] Bhuvan Urgaonkar and Prashant Shenoy. Cataclysm: Handling extreme overloads in Internet applications. In *Proceedings of the International World Wide Web Conference (WWW)*, May 2005.

[42] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the Usenix Annual Technical Conference*, June 2001.

[43] Jianbin Wei and Cheng-Zhong Xu. *e*QoS: Provisioning of client-perceived end-to-end QoS guarantees in web servers. Technical report, Wayne State University, February 2005. Online: http://www.cic.eng.wayne.edu/∼jbwei/self-tuning-fuzzy.pdf.

[44] Matt Welsh and David Culler. Adaptive overload control for busy Internet servers. In *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.