Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers[1]

Chenyang Lu Ying Lu Tarek F. Abdelzaher John A. Stankovic Sang H. Son

Abstract

This paper presents the design and implementation of an adaptive web server architecture to provide relative and absolute connection delay guarantees for different service classes. The first contribution of this paper is an adaptive architecture based on feedback control loops that enforce desired connection delays via dynamic connection scheduling and process reallocation. The second contribution is the use of control theoretic techniques to model and design the feedback loops with desired dynamic performance. In contrast to heuristics-based approaches that rely on laborious hand-tuning and testing iteration, the control theoretic approach enables systematic design of an adaptive web server with established analytical methods. The adaptive architecture has been implemented by modifying an Apache server. Experimental results demonstrate that the adaptive server provides robust delay guarantees even when workload varies significantly.

Keywords: Web server, Quality of Service, feedback control, proportional differentiated service.

I. INTRODUCTION

The increasing diversity of applications supported by the World Wide Web and the increasing popularity of time-critical web-based applications (such as online trading) motivates building QoS-aware web servers. Such servers customize their performance attributes depending on the class of the served requests so that more important requests receive better service. From the perspective of the requesting clients, the most visible service performance attribute is typically the service delay. Different requests may have different tolerances to service delay. For example, one can argue that stock trading requests should be served more promptly than recreational browsing requests. Similarly, in-

[1] This work is funded by NSF grants CCR-9901706 and CNS-0448554.

teractive clients should be served more promptly than background software agents such as web crawlers and prefetching proxies. Some businesses may also want to provide different service delays to different classes of customers (e.g., depending on their monthly fees).

Relative and *absolute* delay guarantees are two common models for service differentiation in web servers. The absolute delay guarantee requires that requests of each class be served within their configured per-class delays if the server is not overloaded. If that is not possible, admission control should be imposed on classes in some pre-defined priority order. Alternatively, delay guarantees could be relaxed for classes in that priority order. In the relative delay guarantee model, a fixed ratio between the delays seen by the different service classes is enforced.

A key challenge in guaranteeing absolute or relative service delays in a web server is that the resource allocation that achieves the desired delay or delay differentiation depends on load conditions that are unknown *a priori*. An important reason for the limited support for differentiated services on current web servers is the lack of robust solutions for enforcing the desired guarantees in face of unpredictable workloads. A main contribution of this paper is the introduction of a feedback control architecture for adapting resource allocation such that the desired delay differentiation between classes is achieved. We formulate the adaptive resource allocation problem as one of feedback control and apply feedback control theory to develop the resource allocation algorithm. We target our architecture specifically for the HTTP 1.1 protocol [20], the most recent version of HTTP that has been adopted at present by most web servers and browsers.

The rest of this paper is organized as follows. In Section II, we define the semantics of delay differentiation guarantees on web servers. The design of the adaptive server architecture to satisfy the delay guarantees is described in Section III. In Section IV, we apply feedback control theory to systematically design a controller to satisfy the desired performance of the web server. The implementation of the architecture on an Apache server and experimental results are presented in Sections V and VI, respectively. We conclude the paper after summarizing related works in Section VII.

II. SERVICE DELAY GUARANTEES ON WEB SERVERS

In this section, we first discuss various components of delays in web services, and then formally specify several models of service delay guarantees on web servers.

A. *Delays in Web Services*

In this paper, we assume a multi-process server model with a finite pool of processes. That is also the model used by the Apache server, the most commonly used web server today. Each server process can only handle one TCP connection at any time instant. Servicing a web request starts when the client's connection request (the SYN packet) is queued on the server's well-known TCP port. The TCP three-way handshake follows. The client then sends an HTTP request to the server process over the TCP connection. The server handles the request and generates a response, which is then sent to the client. Upon the completion of response transmission, the server behaves differently depending on the version of HTTP protocol. If the request is an HTTP 1.0 request, the server process immediately closes the TCP connection. This connection-per-request model results in a large number of short-lived TCP connections and increases overhead. To remedy this problem HTTP 1.1 features *persistent connections* [20], which allow multiple requests to reuse the same connection. Specifically, after a response has been transmitted, the TCP connection is left open in anticipation that the same connection can be reused by a following HTTP request from the same client. If a new HTTP request arrives within a specific interval, the connection is kept open; otherwise it is closed. We focus on HTTP 1.1 since it is the most commonly used HTTP protocol today.

From a client's perspective, the end-to-end delay of a web service includes the *communication delay* on the Internet, the *connection delay* on the server, and the *processing delay* on the request. Of these, server-side delays (as opposed to the network delays) often contribute a significant portion of the end-to-end delay. Controlling the connection delay is especially important in a server running the HTTP 1.1 protocol because a server process remains tied up with a persistent connection even when it is not processing any requests. The operating system typically imposes a limit on the maximum number of concurrent server processes created to prevent thrashing-related performance degradation. This makes the allocation of available processes among classes a very effective means of delay differentiation. The main contribution of this work is thus to develop a novel server process allocation mechanism to support connection delay control in HTTP 1.1 servers via a control-theoretic approach. We note that our work on connection delay control is complimentary to earlier research that focuses on controlling processing delays [1][15][22] and network delays [17].

B. *Semantics of Service Delay Guarantees*

Suppose every HTTP request belongs to a class k ($0 \leq k < N$). The connection delay $C_k(m)$ of class k at the m^{th} sampling instant is defined as the average connection delay of all connections of class k that are established within the time interval $((m\text{-}1)S, mS)$, where S is a constant sampling period. Connection delay guarantees are defined as follows. For simplicity of presentation, we use delay to refer to connection delay in the rest of this paper.

Relative Delay Guarantee: A *desired relative delay* W_k is assigned to each class k. A *relative delay guarantee* $\{W_k \mid 0 \leq k < N\}$ requires that $C_j(m)/C_l(m) = W_j/W_l$ for any class j and l ($j \neq l$)[2]. For example, if class 0 has a desired relative delay of 1.0, and class 1 has a desired relative delay of 2.0, it is required that the connection delay of class 0 should be half of that of class l.

Absolute Delay Guarantee: A *desired absolute delay* W_k is assigned to each class k. An *absolute delay guarantee* $\{W_k \mid 0 \leq k < N\}$ requires that $C_j(m) \leq W_j$ for any class j if there exists a lower priority class $l > j$ and $C_l(m) \leq W_l$ (a lower class number means a higher priority). Since system load can grow arbitrarily high in a web server, it is impossible to satisfy the desired delay of all classes under overload conditions. The absolute delay guarantee requires that all classes receive satisfactory delays

[2] The relative delay guarantee is useful only if there are requests from both classes. If one class stops generating requests, the corresponding relative delay constraint is considered to be satisfied by default.

if the server is not overloaded; otherwise low priority classes suffer guarantee violation earlier than high priority classes.

III. A FEEDBACK CONTROL ARCHITECTURE FOR WEB SERVER QOS

In this section, we present an adaptive web server architecture (as illustrated in Figure 1) to provide the above delay guarantees. A key feature of this architecture is the use of feedback control loops to enforce desired delays via dynamic reallocation of server processes. We describe the design of the components in the following subsections.

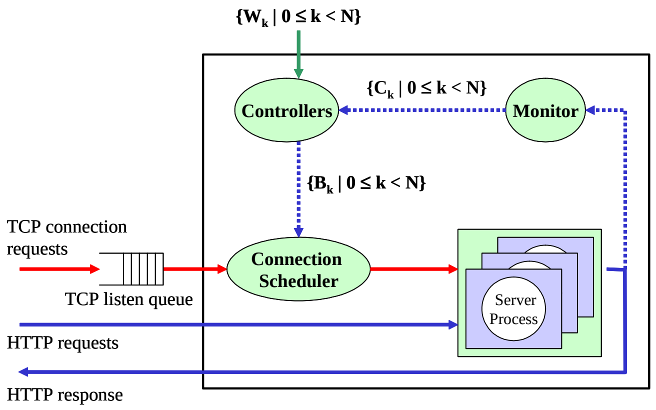


Figure 1: The Feedback Control Architecture for Delay Guarantees

A. Connection Scheduler

The connection scheduler serves as an actuator for controlling the delays of different classes. It listens to the well-known port, accepts every incoming TCP connection request, and uses an adaptive proportional share policy to allocate server processes to handle TCP connections from different classes. At every sampling instant m, every class k ($0 \le k < N$) is assigned a *process budget*, $B_k(m)$. The connections from class k should be served by at most $B_k(m)$ server processes at any time instant in the m^{th} sampling period. For a system with absolute delay guarantees, the total budgets of all classes may exceed the total number of server processes at overload. In this case, the process budgets are satisfied in priority order. Low-priority classes that do not receive enough processes must exercise admission control or simply violate the delay specified for their class. A minimum number of processes can be assigned for such classes to prevent starvation. The connection scheduler controls the order of delay violations (or admission control intervention) at overload. For a server with relative delay guarantees, our relative delay controllers always ensure that processes are allocated in a way that satisfies the guarantee.

The connection scheduler classifies each connection based on certain criteria. For example, the server may classify a connection according to the IP address of the client or the destination (e.g., when multiple virtual servers are hosted on a same physical server). Other possible criteria include HTTP cookies, Browser plug-ins, URL request type and filename path [8]. An advantage of IP-based classification is that it does not require the connection scheduler to read the HTTP header or payload in order to classify a connection. In practice, the classification criteria depend on the application's service level agreements.

For each class k, the connection scheduler maintains a FIFO queue Q_k and a process counter R_k. The queue Q_k holds connections of class k before they are allocated server processes. The counter R_k is the number of processes allocated to class k. After an incoming connection is accepted, the connection scheduler classifies the new connection and inserts the connection descriptor in the scheduling queue corresponding to its class. Whenever a server process becomes available, a connection at the front of a scheduling queue Q_k is dispatched if class k has the highest priority among all eligible classes $\{j \mid R_j < B_j(m)\}$.

For the above scheduling algorithm, a key issue is how to decide the process budgets $\{B_k(m) \mid 0 \le k < N\}$ to achieve the desired relative or absolute delays $\{W_k \mid 0 \le k < N\}$. Note that static mappings from the desired relative or absolute delays to the process budgets cannot work well when the workloads are unpredictable and vary at run time. This problem motivates the use of feedback controllers to dynamically adjust the process budgets to maintain desired delays.

Because the controller may dynamically change the process budgets, a situation can occur when a class *k*'s new process budget $B_k(m)$ exceeds the total number of free server processes and processes already allocated to class *k*. Such a class is called an *under-budget* class. Two different policies, *preemptive* vs. *non-preemptive* scheduling, can be supported in this case. In the preemptive scheduling model, the connection scheduler immediately forces server processes to close the connections of *over-budget* classes whose new process budgets are lower than the number of processes currently allocated to them. A disadvantage of the preemptive model is that it may cause jittery delay in preempted classes because they may have to re-establish connections with the server in the middle of loading a web page. Preempting an existing connection may also introduce unnecessary overhead when most web requests are short-lived. In the non-preemptive scheduling model, the connection scheduler waits for server processes to voluntarily release connections of over-budget classes before it allocates enough processes to under-budget classes. Non-preemptive scheduling may be particularly desirable in commercial web servers where it is important to keep established connections alive in order to conclude the commercial transactions between the customers and the seller [13]. Therefore, our current server only implements the non-preemptive model.

B. Server Processes

The second component of the architecture is a fixed pool of server processes. A server process reads connection descriptors from the connection scheduler. Once a server process closes a connection it notifies the connection scheduler and becomes available for new connections.

C. Monitor

The monitor is invoked at each sampling instant *m*. It computes the average connection delay, $C_k(m)$ $(0 \le k < N)$, of each class *k* during the last sampling period. The sampled connection delays are used by the controller to compute new process budgets.

D. Controllers

The architecture uses one controller for each relative or absolute delay constraint. At each sampling instant m, the controllers compare the sampled connection delays $\{C_k(m) \mid 0 \le k < N\}$ to the desired relative or absolute delays $\{W_k \mid 0 \le k < N\}$, and compute new process budgets $\{B_k(m) \mid 0 \le k < N\}$, which are used by the connection scheduler to reallocate server processes.

Absolute Delay Controllers. The absolute delay of every class k is controlled by a separate absolute delay controller CA_k. The key parameters and variables of CA_k, are shown blow.

Reference VS_k	Desired delay of class k, $VS_k = W_k$.
Output $V_k(m)$	Measured delay of class k, $V_k(m) = C_k(m)$.
Error $E_k(m)$	Difference between the reference and the output, $E_k(m) = VS_k – V_k(m)$.
Control input $U_k(m)$	Process budget $B_k(m)$ of class k.

Table 1: Variables of an absolute delay controller CA_k

The goal of the absolute delay controller, CA_k, is to reduce the error $E_k(m)$ to 0 and hence achieve the desired delay for class k. At every sampling instant m, the absolute delay controller computes the control input using Proportional-Integral (PI) control [21]. PI control has been widely adopted in industry control systems. An important advantage of PI control is that it can often provide robust control performance despite considerable modeling errors [21]. A digital form of the PI control function is

$$U_k(m) = U_k(m\text{-}1) + g(E_k(m) - rE_k(m\text{-}1)) \quad (1)$$

where g and r are design parameters called the *controller gain* and the *controller zero*, respectively. The performance of the web server depends on the values of the parameters. We apply control theory to tune the parameters to achieve desired performance (see Section IV).

For a system with N service classes, the absolute delay guarantees are enforced by N absolute delay controllers $\{CA_k \mid 0 \le k < N\}$. At each sampling instant m, controller CA_k computes the process budget of class k, then allocates these budgets to classes in priority order until no processes are left. Processes are reassigned in a non-preemptive fashion as discussed earlier. A server can avoid starving a low priority class by reserving a minimum number of server processes for that class. Whether a server should implement this policy depends on application requirements.

When the server is severely overloaded, the sum of the process budgets may be higher than the total number of processes available, a situation called control *saturation*. The PI control function in Equation (1) can be modified as follows to prevent the system from becoming unstable after prolonged periods of saturation. If saturation occurred in the previous sampling period, the controller uses the *actual* number of processes allocated to class k in the previous sampling period instead of the last control input $U_k(m\text{-}1)$ to compute the new control input $U_k(m)$.

When the server is underloaded, the sum of the process budgets may be lower than the total number of processes available. If a sudden burst of requests (e.g., a flash crowd) arrive at this time, our server may unnecessarily restrict requests from being assigned to free processes in order to honor the budgets. To deal with this problem a server may allow a service class to "borrow" the free processes for incoming connections even after it has exhausted its own process budget.

Relative Delay Controllers. The relative delay of every two adjacent classes k and k-1 is controlled by a relative delay controller CR_k. CR_k exports the following key parameters and variables (for simplicity we use the same notations for the corresponding parameters and variables of an absolute delay controller and a relative delay controller).

Reference VS_k	Desired delay ratio between class k and k-1, $VS_k = W_k / W_{k-1}$.
Output $V_k(m)$	Measured delay ratio between class k and k-1, $V_k(m) = C_k(m) / C_{k-1}(m)$.
Error $E_k(m)$	Difference between the reference and the output, $E_k(m) = VS_k – V_k(m)$.
Control input $U_k(m)$	*Process ratio*: The ratio between the process budgets of classes k-1 and k, i.e., $U_k(m) = B_{k-1}(m) / B_k(m)$.

Table 2: Variables of a relative delay controller CR_k

The goal of the controller CR_k is to reduce the error $E_k(m)$ to 0 and hence achieve the correct delay ratio between class k and k-1. Similar to the absolute delay controller, the relative delay controller also uses PI control, see Equation (1), to compute the control input (note that the parameters and variables are interpreted differently in an absolute delay controller and a relative delay controller).

For a system with N service classes and M server processes, the relative delay guarantees are enforced by N-1 relative delay controllers $\{CR_k \mid 1 \leq k < N\}$. At every sampling instant m, the system calculates the process budget $B_k(m)$ of each class k as follows.

```
NB_k(m): normalized process budget of class k relative to class N-1. NB_k(m) = B_k(m)/B_{N-1}(m).
sum: the sum of the normalized process budgets of all classes.
M: the total number of server processes.

control_relative_delay
begin
    $NB_{N-1}(m) = 1$; $Sum = 1$;
    for ( $k = N-2$; $k \geq 0$; $k$-- ) {
        Call controller $CR_{k+1}$ to get the process ratio $U_{k+1}(m)$ between classes $k$ and $k+1$;
        $NB_k(m) = NB_{k+1}(m)U_{k+1}(m)$;
        $sum = sum + NB_k(m)$;
    }
    for ( $k = N-1$; $k \geq 0$; $k$-- )
        $B_k(m) = M * NB_k(m)/sum$;
end
```

In the rest of the paper, we use the term *closed-loop server* to refer to a server instrumented with feedback controllers, and the term *open-loop server* to refer to a server without the controllers.

IV. DESIGN OF THE CONTROLLERS

We apply a control-theoretic methodology to design our controllers. We first specify the performance requirements for the controllers, and then use system identification techniques to establish dynamic models for the web server. Based on the dynamic models, we use the Root Locus method to design controllers that meet the performance specifications.

A. Performance Specifications

We adopt a set of metrics from feedback control theory [21] to characterize the dynamic performance of a web server.

- **Stability**: A stable system should have bounded output in response to bounded input. A stable relative delay controller ensures that the delay ratio remains bounded at run-time. Since it is impossible to always achieve bounded absolute delay in face of arbitrary workload, we relax the stability requirement for absolute delay control. A stable absolute delay controller ensures that the absolute delay is bounded if the server has enough processes to satisfy the process budgets computed by the controller. Stability is a necessary condition for achieving the desired delay guarantees.

- **Settling time**: The time it takes the system output to converge to the vicinity of the reference and enter a steady state. The settling time represents the efficiency of the controller, i.e., how fast the server can converge to the desired relative or absolute delays.
- **Steady state error**: The difference between the reference and average of system output in the steady state. The steady state error represents the accuracy of the controllers in enforcing the desired delays. A low steady state error indicates that the web server provides desired delay guarantees in a steady state.

B. System Identification

A dynamic model describes the mathematical relationship between the control input and the output of a system using difference equations. It provides a foundation for the design of the controller. Since server queues are integrators of flow (which gives rise to difference equations) the controlled server system can be modeled as a difference equation with unknown parameters. System identification [6] is used to estimate parameter values. We now describe the components used for system identification. For notational simplicity, we omit the class index k from expressions such as $U_k(m)$ and $V_k(m)$ in the rest of this section. The web server is modeled as a difference equation as follows:

$$V(m) = \sum_{j=1}^{n} a_j V(m-j) + \sum_{j=1}^{n} b_j U(m-j) \qquad (2)$$

In an n^{th} order model, there are $2n$ parameters $\{a_j, b_j \mid 1 \le j \le n\}$ that need to be decided by the least-squares estimator. The difference equation model reflects the fact that the output of an open-loop server depends on previous inputs and outputs (i.e., delays are correlated and depend on the recent process allocation history). Intuitively, the dynamics of a web server are due to the queuing of connections and the non-preemptive scheduling mechanism. The connection delay may depend on the number of server processes allocated to its class in several previous sampling periods. Furthermore, when class *k*'s process budget is increased, the non-preemptive connection scheduler may have to wait for server processes of other classes to close their connections in order to reclaim enough processes to class *k*.

To stimulate the dynamics of the open-loop server, we use a pseudo-random digital white noise generator to randomly switch two classes' process budgets between two input values. The input values to the white noise generator are selected based on the estimated range of the control inputs (the process ratio or process budget) at run time. White noise input has been commonly used for system identification. A standard algorithm can be found in [6].

The least squares estimator is the key component of the system identification. In this section, we review its mathematical formulation and describe its use to estimate the model parameters. The derivation of estimator equations is given in [6]. The estimator is invoked periodically at every sampling instant. At the m^{th} sampling instant, it takes as input the current output $V(m)$, n previous outputs $V(m-j)$ $(1 \le j \le n)$, and n previous inputs $U(m-j)$ $(1 \le j \le n)$. The measured output $V(m)$ is fit to the model described in Equation (2). We define the vector $q(m) = (V(m-1) \ldots V(m-n)\ U(m-1) \ldots U(m-n))^{T}$, and the vector $\theta(m) = (a_1(m) \ldots a_n(m)\ b_1(m) \ldots\ b_n(m))^{T}$, which describes the estimates of the model parameters in Equation (2). These estimates are initialized to 1 at the start of the estimation. Let $R(m)$ be a square matrix whose initial value is set to a diagonal matrix with the diagonal elements set to 10. The estimator's equations at sampling instant m are as follows [6]:

$$\gamma(m) = (q(m)^{T} R(m-1) q(m) + 1)^{-1} \qquad (3)$$

$$\theta(m) = \theta(m-1) + R(m-1) q(m) \gamma(m) (V(m) - q(m)^{T} \theta(m-1)) \qquad (4)$$

$$R(m) = R(m-1)(I - q(m)\gamma(m) q(m)^{T} R(m-1)) \qquad (5)$$

At any sampling instant, the estimator can "predict" a value $V^*(m)$ of the output by substituting the current estimates $\theta(m)$ into Equation (2). The difference $V(m)$-$V^*(m)$ between the measured output and the prediction is the estimation error. It can be proved that the least squares estimator iteratively updates the parameter estimates such that $\sum_{0\le i\le m}(V(i) - V^*(i))^2$ is minimized.

As presented in Section VI.A., our empirical results showed that the controlled system can be modeled as a second order difference equation:

$$V(m) = a_1V(m\text{-}1) + a_2V(m\text{-}2) + b_1U(m\text{-}1) + b_2U(m\text{-}2) \quad (6)$$

For absolute delay control, the system output (absolute delay) is not 0 when the system input (the process budget) remains at 0. Since this property is inconsistent with the above model structure, we linearize the model to fit our model structure by feeding the difference between two consecutive inputs ($B_0(m)$ - $B_0(m$-1)) and the difference between two consecutive outputs ($C_0(m)$ - $C_0(m$-1)) to the least squares estimator to estimate the model parameters. Such linearization is unnecessary for relative delay control as it does not have the above bias problem.

C. Root-Locus Design

We apply the Root Locus [21] method to design the controllers. The controlled system model (Equation (6)) can be converted to a transfer function $G(z)$ from the control input $U(z)$ to the output $V(z)$ in the z-domain, given by Equation (7) below. The PI controller (Equation (1)) can be converted to a transfer function from the error $E(z)$ to the control input $U(z)$ in the z-domain, given by Equation (8). Given the controlled system model and the controller model, the closed-loop system can be modeled as a transfer function from the reference $VS(z)$ to the output $V(z)$, given by Equation (9).

$$G(z) = \frac{V(z)}{U(z)} = \frac{b_1 z + b_2}{z^2 - a_1 z - a_2} \quad (7)$$

$$D(z) = \frac{U(z)}{E(z)} = \frac{g(z-r)}{z-1} \quad (8)$$

$$G_c(z) = \frac{V(z)}{VS(z)} = \frac{D(z)G(z)}{1 + D(z)G(z)} \quad (9)$$

The roots of the denominator polynomial of a transfer function are called its *poles*. The closed-loop system is stable when all the poles of its transfer function (Equation (9)) are placed inside the unit circle |z|=1 and unstable when any pole is placed outside the unit circle. In addition, we need to place them at the appropriate positions in the unit circle in order to satisfy the performance specifications, such as ensuring a desired settling time to achieve quick response. The Root Locus is a graphical technique that plots the traces of poles of a closed-loop system on the *z*-plane as its controller parameters change. Control textbooks and software tools such as MATLAB typically contain maps of the z-plane that plot the contours of locations of poles that correspond to given transient response parameter values such as settling time. In practice, these maps are what an engineer could use to locate the poles such that settling time requirements are satisfied. For example, the settling time depends on the magnitude of the poles. The closer the poles are to the origin, the faster the convergence and the shorter the settling time. However, being too close to the origin may cause other unwanted side effects such as reduced robustness to modeling errors. We thus choose to place the poles within the unit circle while satisfying the required setting time. Due to space limitations, we only summarize results of the design in this paper. The Root Locus method can be found in control textbooks such as [21].

To design the relative delay controller, we first estimate the model parameters in Equation (6) through system identification experiments using Workload A described in Section VI.A. We then use the Root Locus tool in MATLAB to place the closed-loop poles for relative delay control at (0.70, 0.38±0.62*i*) by setting the relative delay controller’s parameters to *g* = 0.30 and *r* = 0.05. Similarly, to design the absolute delay controller, we first estimate the model parameters in Equation (6) through system identification experiments using Workload A' described in Section VI.A, and then place the closed-loop poles for absolute delay control at (0.53, -0.16±0.51*i*) by setting the absolute delay controller’s parameters to *g* = -0.80, *r* = 0.30.

D. Control Analysis

Our analysis based on linear control theory shows that the closed-loop server with above pole placement has the following properties.

Stability: The closed-loop systems with the relative and the absolute delay controllers are stable because all the closed-loop poles are in the unit circle, i.e., $|p_j| < 1$ ($0 \leq j \leq 2$).

Settling time: The positions of the closed-loop poles ensure that the relative delay controller can achieve a settling time around 270 seconds, and the absolute delay controller a settling time of 150 seconds.

Steady state error: Both the relative delay controller and the absolute delay controller achieve zero steady state error. This result can be proved using the Final Value Theorem in linear control theory [21]. Observe that the final value theorem holds true regardless of the exact values of model parameters due to the presence of integral control action in the system (namely, in the PI controller). Integral control (the I in the PI controller) simply increases resource allocation as long as the delay is too high and decreases the allocation when the delay is too low. Hence, it always acts in the direction that reduces the error to zero, thereby eliminating steady-state error. In other words, the steady state error should be zero even if the model is not accurate as long as the run-time system does not run into an actuator saturation limit. This result means that the closed-loop system with our controllers can, on average, achieve the desired relative and absolute delays.

D. Limitations of Control Design and Analysis

Modeling. The models estimated via system identification are approximations of the system dynamics. If the deployed system configuration or workload range deviates significantly from those used for system identification, the estimated system model will become inaccurate. Such modeling errors may cause the system properties (e.g., the setting time) to deviate from our analytical results. However, feedback control systems can usually tolerate a certain degree of modeling inaccuracy. Indeed, the robustness of simple controllers such as PI and PID with respect to modeling errors is the primary reason for their proliferation in industrial applications. We verify the robustness of our controller design in face of system variations through experiments described in Section VI. Moreover, we can further improve the robustness of the server by developing *adaptive* controllers [6][29] that automatically adjust its control parameters based on the results of online system identification.

Control analysis. It should be noted that the validity of our control analysis is affected by the accuracy of *linear* system models used in the analysis. In particular, our models ignore several *non-linear* properties of real-world web servers. For instance, the process budgets are subject to the constraint that the budget must be between zero and the maximum number of server processes available in the system. In addition, the bursty nature of web workloads can cause a significant noise in system output. Therefore the delays on our servers may not track the reference precisely in each sampling period. Instead, they fluctuate while the *average* delays remain close to the references as shown in our experimental results presented in Section IV.

V. IMPLEMENTATION

We modified the source code of Apache 1.3.9 [5] and added a new library that implemented a connection manager (including the connection scheduler, the monitor and the controllers). The server was written in C and tested on a Linux platform. The server is composed of a connection manager process and a fixed pool of server processes. The connection manager process communicates with each server process with a separate UNIX domain socket. The connection manager runs a loop that listens to the web server's TCP socket and accepts incoming connection requests. In our experiments, each connection request is classified based on its sender's IP address and scheduled by a connection scheduler function. The connection scheduler dispatches a connection by sending its descriptor to a free server process through its UNIX domain socket. The connection manager time-stamps the acceptance and dispatching of each connection. The difference between the acceptance and the dispatching time is recorded as the connection delay. Strictly speaking, the connection delay should also include the queuing time in the TCP listen queue in the kernel. However, the kernel delay is negligible in this case because the connection manager always greedily accepts (dequeues) all incoming TCP connection requests in a tight loop. The monitor and the controllers are invoked periodically at every sampling instance. For each invocation, the monitor computes the average delay for each class. This information is passed to the controller, which then computes new process budgets.

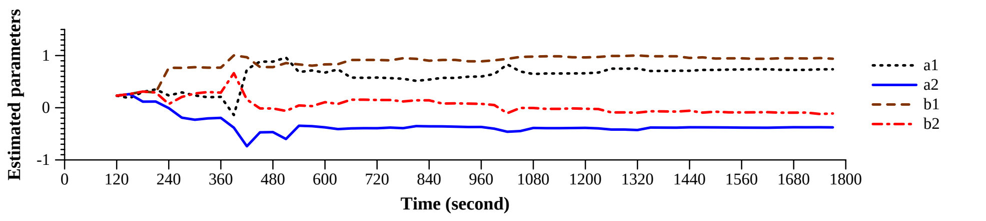
We modified the server processes so that they accept connection descriptors from UNIX domain sockets (instead of the common TCP listen socket). When a server process closes a connection, it notifies the connection manager through the UNIX domain socket. The server can be configured as a closed-loop/open-loop server by turning on/off the controllers. An open-loop server can be configured for system identification or as a baseline for performance evaluation.

VI. EXPERIMENTATION

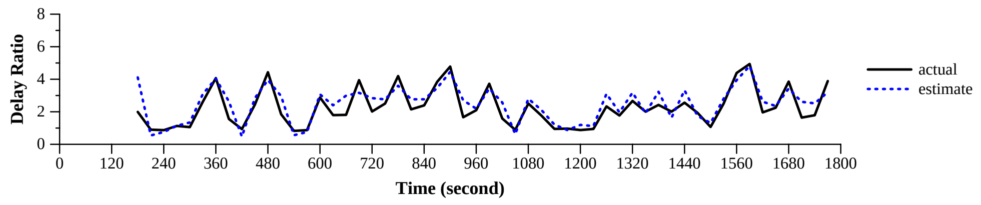
All experiments were conducted on a testbed of Linux PCs connected via a 100 Mbps Ethernet. Each PC had a 450MHz AMD K6-2 processor and 256 MB RAM. One PC was used to run the web server with HTTP 1.1, and up to four other PCs were used to simulate clients that stress the server with a synthetic workload. The experimental setup was as follows.

Client: We used SURGE [9] to generate HTTP requests to the server. SURGE uses a number of *user equivalents* (also called *users* for simplicity) to emulate the behavior of real-world clients. The load on the server can be adjusted by changing the number of users on the client machines. Up to 600 concurrent users were used in our experiments.

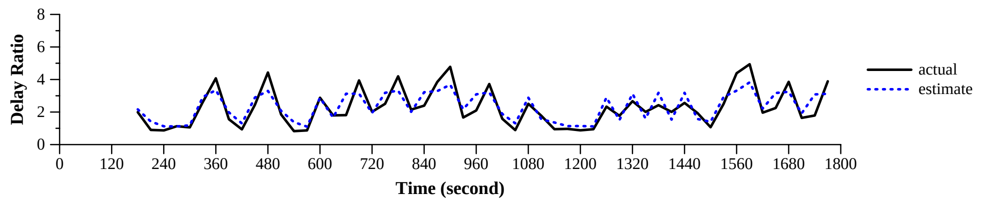
Server: The total number of server processes was configured to 128. Since service differentiation is most necessary when the server is overloaded, we set up the experiment such that the ratio between the number of users and the number of server processes could drive the server to overload. Note that although large web servers such as on-line trading servers usually have more server processes, they also tend to have many more users than the workload we generated. Therefore, our configuration can be viewed as an emulation of real-world overload scenarios at a smaller scale. The connection time-out was 15 seconds (the default value in Apache 1.3.9).



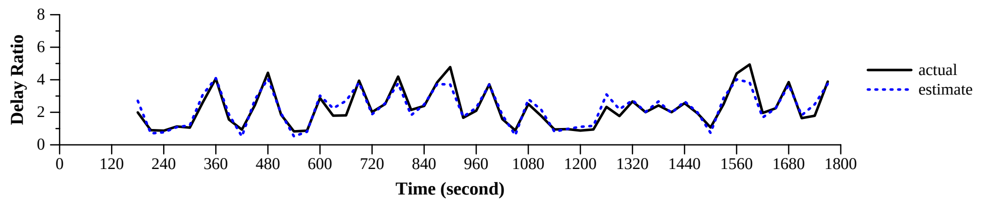
(a) Estimated model parameters (second order model)



(b) Modeling error (second-order model)



(c) Modeling error (first-order model)



(d) Modeling error (third-order model)

Figure 2: System Identification for Relative Delay Control (Workload A)

A. System Identification

We now present the results of our system identification experiments.

1) Relative Delay

In the first set of experiments, we use four client machines to generate the workload from two classes of users. The process ratio $U(m) = B_0(m)/B_1(m)$ is initialized to 1. At each sampling instant, the white noise randomly toggles the process ratio between 3 and 1. The process ratio inputs are chosen based on the estimated range of fluctuation in the control input (process ratio) at run time. The measured relative delay $V(m) = C_1(m)/C_0(m)$ is fed to the least squares estimator to estimate the model parameters in Equation (2). To evaluate the sensitivity of the model parameters to workloads, we carried out system identification experiments using three workloads with different user populations: Workload A comprised of 200 class 0 users and 200 class 1 users; Workload B comprised of 150 users class 0 users and 250 class 1 users; and Workload C comprised of 300 class 0 users and 300 class 1 users.

We first study the results with Workload A. Figure 2(a) shows the estimated parameters of the second order model in Equation (6) at successive sampling instants in a 30 minutes run. The estimator and the white noise generator are turned on 2 minutes after SURGE started in order to avoid its start-up phase. The estimations of (a_1, a_2, b_1, b_2) converge to (0.74, -0.37, 0.95, -0.12). To verify the accuracy of the model, we change the seed of the white noise to generate a different sequence of process ratios on the server, and compare the *actual* delay ratio to that *predicted* by the estimated model. As shown in Figure 2(b), the predictions of the estimated model are consistent with the actual relative delays. This result indicates that the second-order model is adequate for control design.

We re-run the experiments to estimate a first order model (see Figure 2(c)) and a third order model (see Figure 2(d)). To quantify the model accuracy, we perform residual analysis to compute the variability explained by the models using the data obtained after 210 seconds in each run. The R^2 values of the first order, second order and third order models are 0.72, 0.80, and 0.90, respectively. The results demonstrate that higher order models are more accurate than lower order models. We choose the second order model as a compromise between accuracy and complexity.

The above results are reported for a sampling period of 30 seconds. This sampling period is chosen as a compromise between accuracy and overhead. While longer sampling periods may fail to capture some of the model dynamics, shorter periods may not considerably improve the accuracy of the model. For example, we carry out system identification experiments with several workloads using 10 seconds as the sampling period and verify the accuracy of estimating the system with second order model via residual analysis. The variability R^2 values we obtained for a second order model at 10 seconds is only slightly higher than 0.8. We keep 30 seconds as the sampling period to reduce the control overhead.

We design the controllers based on the model estimated using Workload A. We now analyze the robustness of our control design with respect to variations in model parameters. With Workload B the estimated model parameters (a_1, a_2, b_1, b_2) converge to (0.31, -0.27, 2.28, 0.08) and $R^2 = 0.86$. With Workload C the parameters converge to (0.56, -0.26, 0.47, 0.21) and $R^2 = 0.88$. Control analysis shows that our relative delay controller designed based on Workload A places the poles of the closed-loop system at (0.52, $0.05\pm0.72i$) and (0.71, $0.36\pm0.50i$) when it is applied to the models estimated using Workloads B and C, respectively. As both poles remain within the stability range under both workloads, our controller can theoretically achieve stability and zero steady-state error for all three workloads. We validate our analysis through performance evaluation (see Section VI.B).

2) Absolute Delay

We used three client machines to generate the workloads from two classes. Three workloads with different user populations were generated: (i) Workload A' comprised of 100 class 0 users and 400 class 1 users; (ii) Workload B' comprised of 150 class 0 users and 250 class1 users; and (iii) Workload C' comprised of 200 class0 users and 300 class1 users. The input of the open-loop system is the process budget $U(m) = B_0(m)$ of class 0. $B_0(m)$ is initialized to 25. At each sampling instant, the white noise randomly toggles the process budget between 100 and 25. The output is the delay $V(m) = C_0(m)$ of class 0. With Workload A' the estimations of (a_1, a_2, b_1, b_2) converge to (-0.13, -0.03, -0.82, -0.52). To verify the accuracy of the model, we change the seed of the white noise to generate a different sequence of process budget on the server, and compare the actual difference between two consecutive delay samples with that predicted by the estimated model. Similar to the relative delay case, the prediction of the estimated model is consistent with the actual delay throughout the 30 minutes run and achieves an $R^2 = 0.80$. With Workload B' the estimated model parameters (a_1, a_2, b_1, b_2) converge to (0.14, -0.05, -0.36, -0.15) and R^2=0.92. With Workload C' the estimation converge to (0.25, -0.03, -0.49, -0.25) and R^2=0.86. This result shows that the estimated second order model is adequate for designing the absolute delay controllers. Control analysis shows that our absolute delay controller places the poles of the closed-loop system at (0.71, 0.07±0.34*i*) and (0.48, 0.19±0.38*i*) when it is applied to the models estimated using Workloads B' and C', respectively. Therefore, the absolute delay controller designed based on Workload A' can theoretically achieve stability and zero steady-state error for all three workloads.

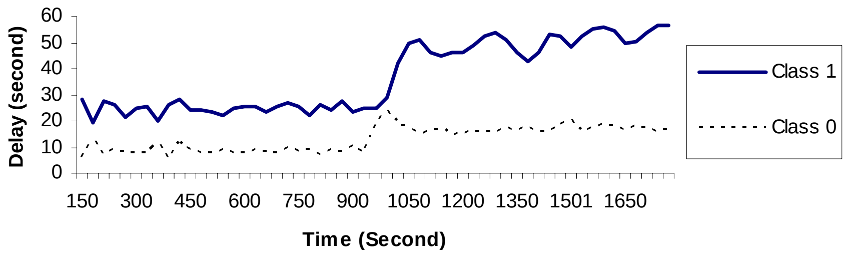
B. Performance Evaluation

We perform four sets of experiments to evaluate the performance of our web server with two or three service classes. Note that we only used two classes for the system identification experiments (see Section VI.A), and used the estimated models to design the controllers for both two and three classes. The empirical results presented in this subsection demonstrate that the models estimated based on two classes are sufficiently accurate for designing robust controllers for three classes.

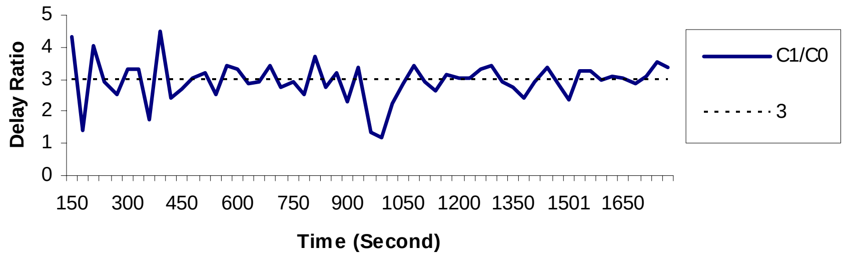
1) Relative Delay Control for Two Classes

To evaluate the relative delay control for two classes, we use three different workload configurations. Workload D is generated by four client machines evenly divided into two classes. Each client machine emulates 100 users. In the first half of each run, only one class 0 client machine and two class 1 client machines generate HTTP requests to the server. The second class 0 machine starts generating HTTP requests 870 seconds later than the other three machines. Therefore, the user population of class 0 changes from 100 to 200 in the middle of each run, while the user population of class 1 remains 100 throughout each run. This workload is designed to stress-test the server’s capability to reallocate server processes as the workload suddenly changes.

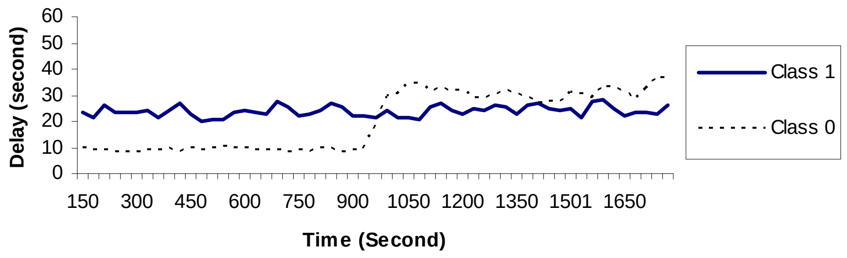
The reference input to the controller is $W_1/W_0 = 3$. The process ratio $B_0(m)/B_1(m)$ is initialized to 1. To avoid the starting phase of SURGE, the controller is turned on 150 seconds after SURGE. An open-loop server is also tested as a baseline. The process allocation in the open-loop server is hand-tuned based on profiling experiments running a same workload as the first half of Workload D.



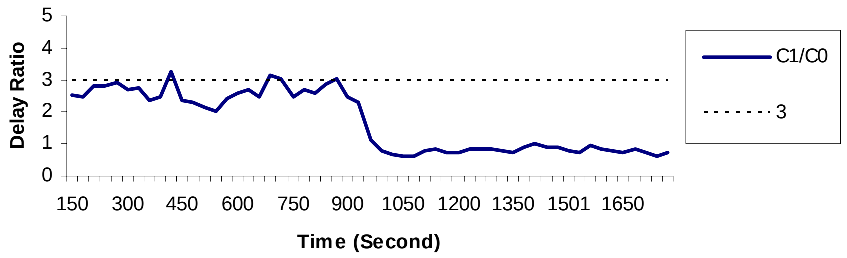
(a) Closed-loop Server: Delay



(b) Closed-loop Server: Delay Ratio



(c) Open-loop Server: Delay



(d) Open-loop Server: Delay Ratio

Figure 3: Experimental Results of Relative Delay Control for Two Classes (Workload D)

Figures 3(a)(b) show the performance of the closed-loop server in a typical run. When the controller is turned on at 150 seconds, the delay ratio $C_1(m)/C_0(m)$ = 28.5/6.5 = 4.4 due to the incorrect initial process allocation. The feedback control loop automatically reallocates processes so that the delay ratio converges to the vicinity of the reference $W_1/W_0 = 3$. After the number of class 0 users suddenly increases from 100 to 200 at 870 seconds, the delay ratio transiently drops to 1.2 at 960 seconds. The feedback control reacts to the load variation by allocating more processes to class 0 while de-allocating processes from class 1. The delay ratio converges again to the vicinity of the set point. As predicted by our analysis, the server remains stable throughout the run. Although the delay ratio oscillates slightly at steady states due to the noise in the workload, the average delay ratio remains close to the reference. The results from the closed-loop server experiments demonstrate that the closed-loop server can (1) *self-tune* its process allocation to achieve desired delay ratios, and (2) maintain *robust* relative delay guarantees despite significant variations in user population. These capabilities are highly desirable in web servers that often face bursty workloads [14]. The experimental results also validate our control design and analysis.

In contrast, as shown in Figures 3(c)(d), while the hand-tuned open-loop server achieves satisfactory relative delays when the workload conforms to its expectation (from 150 to 900 seconds), it severely violates the relative delay guarantee after the workload changes. From 960 seconds to the end of the run, class 0 receives *longer* delays than class 1. Therefore, an open-loop server cannot maintain desired relative delay guarantees in face of varying workload.

To evaluate the control performance under different workload configurations, we re-run the experiments with Workload B comprised of 150 class 0 users and 250 class 1 users. The reference input to the Controller is $W_1/W_0 = 3$. As shown in Figure 4, the controller again successfully keeps the average delay ratio close to the reference with the new workload.

We also evaluate the controller performance with different delay ratio goals. This set of experiments use Workload C comprised of 300 users from both class 0 and class 1. The reference input to the controller is W_1/W_0 = 3, 2, and 4 respectively. As shown in Figures 5(a-c), the controller always keeps the average delay ratio close to the chosen reference. It demonstrates that the designed controller can provide desired performance specified by the reference values.

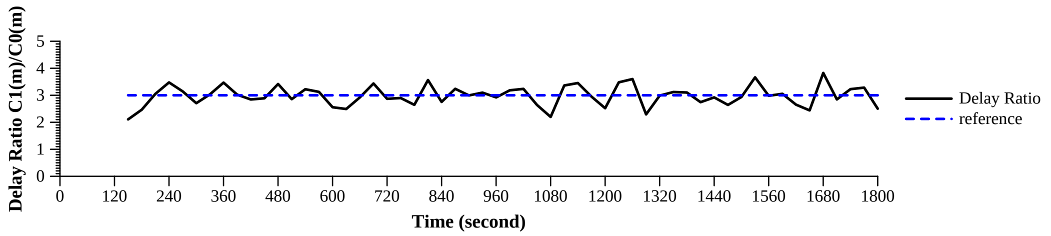


Figure 4: Experiment Results of Relative Delay Control for Two Classes (Workload B)

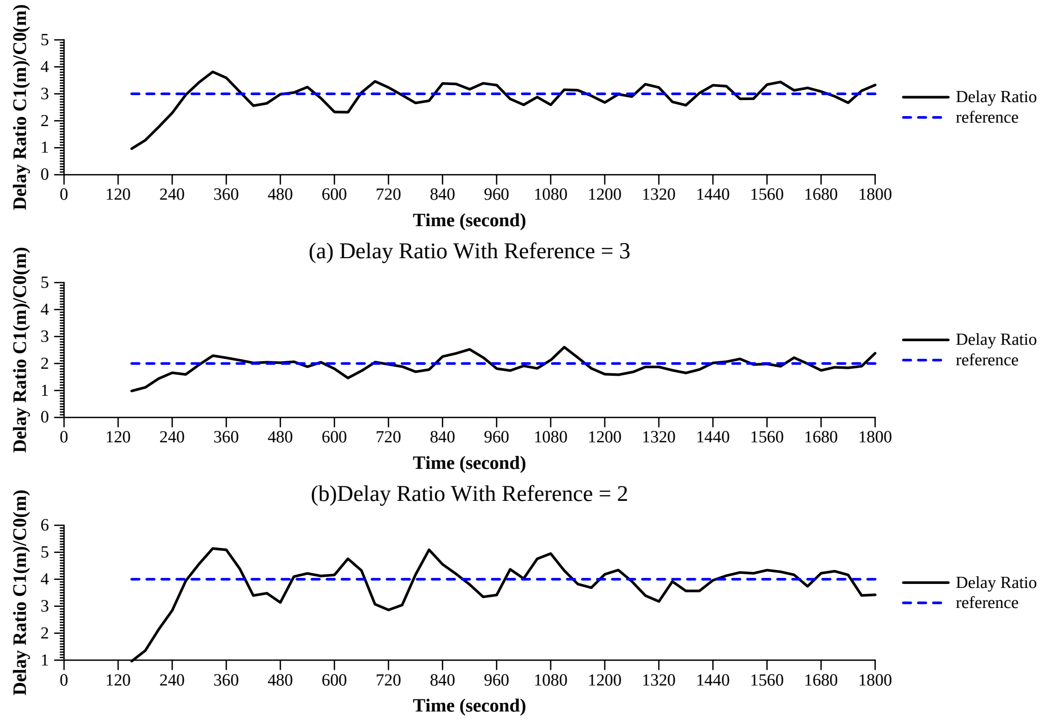
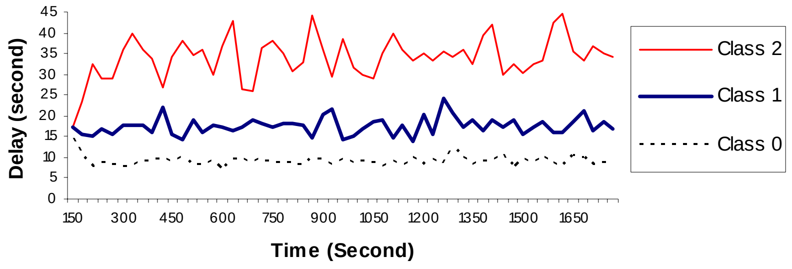


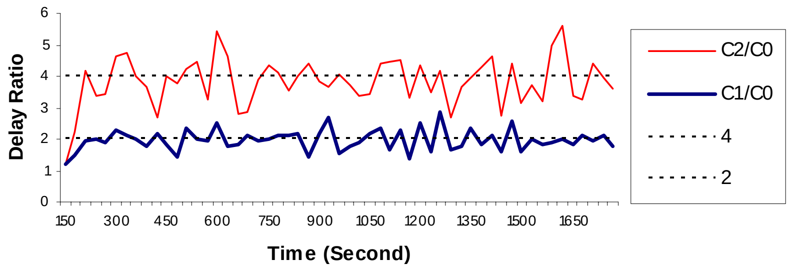
Figure 5: Experiment Results of Relative Delay Control for Two Classes (Workload C)

2) Relative Delay Control for Three Classes

In this experiment, each class has 100 users simulated by a separate client machine. The desired delay ratios are W_0:W_1:W_2 = 1:2:4. The process ratios are initialized to B_0:B_1:B_2 = 1:1:1. The results are shown in Figures 6(a)(b). From the designed settling time (240 seconds after the controller is turned on at 150 seconds) to the end of the run, the average delays of the three classes are 9.0, 17.7, and 34.7 seconds, respectively, while the average delay ratios are C_0:C_1:C_2 = 1.0:2.0:3.9. This experiment demonstrates that our server can effectively control the relative delays of three classes by combining the inputs from two controllers.



(a) Delay



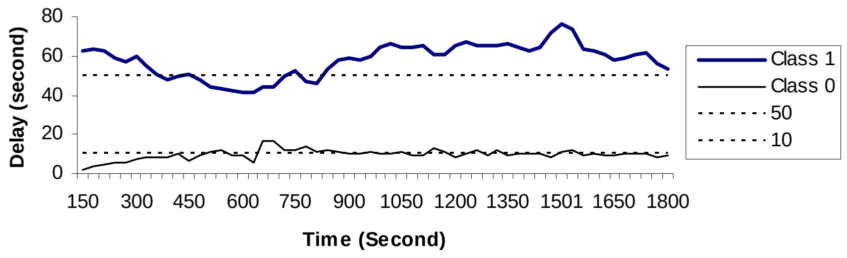
(b) Delay Ratio

Figure 6: Experimental Results of Relative Delay Control for Three Classes

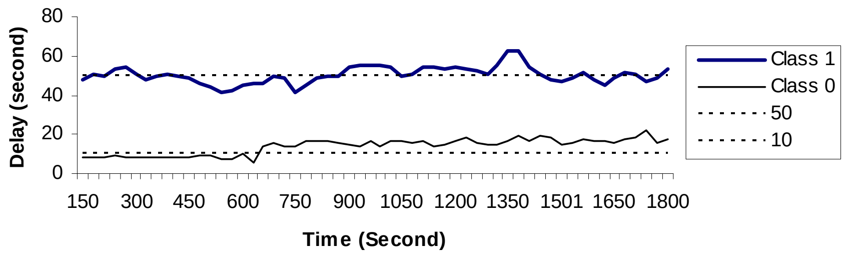
3) Absolute Delay Control for Two Classes

In this set of experiments we use two machines to simulate 400 class 1 users and two other machines to simulate class 0 users. Class 0 has 75 users in the first 660 seconds, and its user population increases to 100 in the middle of each run. The user population of class 1 remains 400 throughout each run. The desired delays for classes 0 and 1 are $(W_0, W_1) = (10, 50)$ seconds. The process budget for each class is initialized to 64 in the closed-loop server. The process budgets in the open-loop server are hand-tuned to achieve the desired absolute delays for the initial workload (comprised of 75 class 0 users and 400 class 1 users) through profiling.

The performance of the closed-loop server in a typical run is shown in Figure 7(a). In the first half of the run, the delays of both classes remain close to their desired delays. However, after the number of class 0 users increases from 75 to 100 at 660 seconds, the delay of class 0 jumps to 16.1 seconds. The controller for class 0 reacts to the violation of its delay guarantee by increasing its process budget. Since all the server processes have been allocated and class 0 has a higher priority, the server fulfills its process budget by taking processes away from class 1. The delay of class 0 settles down to the vicinity of its reference input within the designed settling time (150 seconds) and remains close to the reference throughout the rest of run. Meanwhile, class 1 suffers a longer delay as the server allocates most processes to class 0, opting to violate the delay requirement of the lower priority class (class 1). Note that even though the controller for class 1 increases its process budget in response to the excessive delay, its input is not fulfilled due to the lack of available processes.



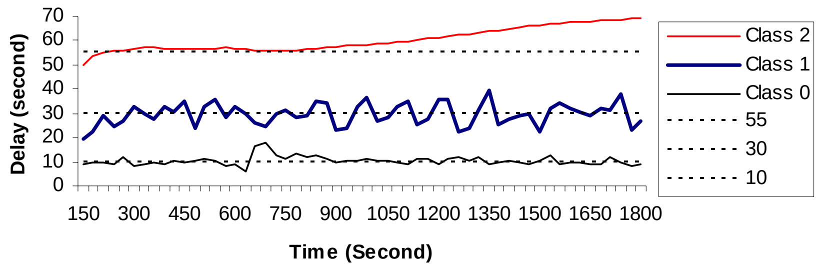
(a) Closed-loop Server



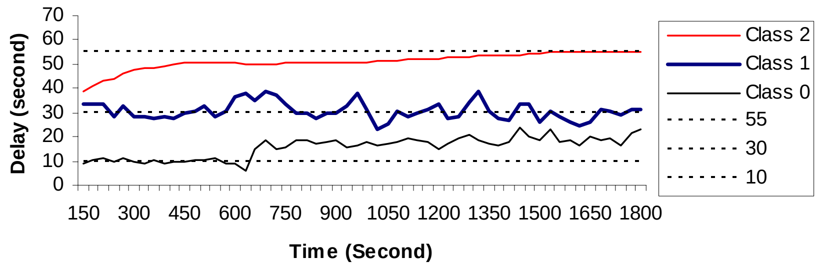
(b) Open-loop Server

Figure 7: Experimental Results of Absolute Delay Control for Two Classes

The performance of the open-loop server in a typical run is shown in Figure 7(b). The open-loop server achieves desired delays for both classes when the workload is the same as expectation. However, after the user population of class 0 changes at 660 seconds, the delay of class 0 increases significantly and violates the desired delay. Note that while both the open loop server and the closed-loop server violate the delay guarantee of one of the service classes, the closed-loop server conforms to the requirement of an absolute delay guarantee (that the desired delay of a low-priority class should be violated earlier than that of a high-priority class).



(a) Closed-loop Server



(b) Open-loop Server

Figure 8: Experimental Results of Absolute Delay Control for Three Classes

4) Absolute Delay Control for Three Classes

In this set of experiments, one machine simulates 300 users from class 2, the second machine simulates 100 users from class 1, and the third machine simulates users from class 0. For the first 660 seconds, only 75 users are simulated from class 0. The user population of class 0 changes from 75 to 100 in the middle of each run, while the user population of class 2 and 1 remain the same throughout each run. The desired delays are (W_0, W_1, W_2) = (10, 30, 55) seconds. The process budget for every class is initialized to around 42 in the closed-loop server. The process budgets in the open-loop server are hand-tuned to achieve the desired delays for the initial workload through profiling. As shown in Figures 8(a)(b), unlike the open loop server, the closed-loop server enforces the absolute delay guarantee by satisfying the required delays of the high-priority classes.

VII. RELATED WORK

Support for different service classes on the web has been investigated in recent literature. For example, Eggert and Heidemann proposed an architecture in which restrictions are imposed on the amount of server resources which are available to basic clients [19]. Bhatti and Friedrich developed a web server architecture that maintains separate service queues for premium and basic clients, thus facilitating their differential treatment [8]. Vasiliou and Lutfiyya proposed another web server architecture and scheduling algorithms for differentiated QoS [37]. Admission control and scheduling algorithms for providing premium clients with better service were also studied in [4][13].

Recent research on server resource management and processor scheduling aimed at optimizing the processing delay and throughout in server systems. Our work on connection delay differentiation is complimentary to earlier research that focuses on processing delays on end-systems [22] and network delays [17]. Several other projects such as [7][18] developed kernel level mechanisms to achieve overload protection and resource isolation in server systems. None of them provided connection delay differentiation in web servers. The integration of our work with those techniques will enable web services to provide *end-to-end* delay guarantees.

Although real-time scheduling may be used to provide certain degrees of delay differentiation, existing scheduling algorithms have several limitations when applied to web servers. Fixed-priority scheduling can provide absolute delay differentiation, but it cannot support *relative* delay guarantees. Scheduling connections using the Earliest Deadline First policy [25] may improve the number of connections that meet their delay requirements, but it does not provide performance isolation for high priority classes in overload situations. Finally, proportional-share scheduling [17][22] can be used to allocate processes to different service classes. However, it is difficult to determine the right shares when the workload properties can vary at run time. Our feedback control approach enhances proportional-share scheduling by automatically adapting the process budgets.

Control-theoretic approaches have been adopted in a number of software systems such as real-time embedded systems [3][12][28], visual tracking [24], database servers [34], and network storage systems [23][27]. A survey on feedback performance control in software services is presented in [2]. Recent work on applying control-theoretic techniques in Internet servers is directly related to this work. Parekh *et al*. developed control-theoretic solutions for maintaining desired queue length in a Lotus email server via admission control [33]. The authors of [1] designed a feedback control loop to enforce desired CPU utilization through content adaptation on a web server. Diao et al. [15] used system identification and multi-input-multi-output control to achieve desired CPU and memory utilization on a web server. The control-theoretic approaches have also been integrated with queueing models to improve the control performance on web servers [30][35]. Neither of those projects can provide both absolute and relative guarantees on connection delays in a web server.

VIII. Conclusion

In this paper, we first present an adaptive server architecture for enforcing desired absolute or relative connection delays via dynamic process reallocation. We then apply a control-theoretic methodology to systematically design the feedback control loop to achieve satisfactory dynamic performance. Finally, we implement and evaluated the adaptive architecture on an Apache web server. Experimental results demonstrate that our adaptive server provides robust delay guarantees even when the workload fluctuates significantly at run time.

IX. References

[1] T. F. Abdelzaher and N. Bhatti, Web Server QoS Management by Adaptive Content Delivery, IWQoS 1999.

[2] T. F. Abdelzaher, J. A. Stankovic, C. Lu, R. Zhang, and Y. Lu, Feedback Performance Control in Software Services, IEEE Control Systems, 23(3), June 2003.

[3] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, Analysis of a Reservation-based Feedback Scheduler, IEEE Real-Time Systems Symposium, 2002.

[4] J. Almedia, M. Dabu, A. Maniknтty, and P. Cao, Providing Differentiated Levels of Service in Web Content Hosting, First Workshop on Internet Server Performance, June 1998.

[5] Apache Software Foundation, http://www.apache.org.

[6] K. J. Astrom and B. Wittenmark, Adaptive Control (2nd Ed.), Addison-Wesley, 1995.

[7] G. Banga, P. Druschel, and J. C. Mogul, Resource Containers: A New Facility for Resource Management in Server Systems, OSDI 1999.

[8] N. Bhatti and R. Friedrich, Web Server Support for Tiered Services. IEEE Network, 13(5), Sept.-Oct. 1999.

[9] P. Barford and M. E. Crovella, Generating Representative Web Workloads for Network and Server Performance Evaluation, SIGMETRICS, 1998.

[10] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, An Architecture for Differentiated Services, IETF RFC 2475, 1998.

[11] A. Bouch, N. Bhatti, and A. J. Kuchinsky, Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service, ACM CHI'2000. April 2000.

[12] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén, Feedback-Feedforward Scheduling of LQG-Control Tasks, Real-time Systems, 23(1/2), 2002.

[13] L. Cherkasova and P. Phaal, Peak Load Management for Commercial Web Servers Using Adaptive Session-Based Admission Control, Proc. 34th Hawaii International Conference on System Sciences, January 2001.

[14] M. E. Crovella and A. Bestavros, Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes, IEEE/ACM Transactions on Networking, 5(6): 835-846, December 1997.

[15] Y. Diao, N. Gandhi, J.L. Hellerstein, S. Parekh, and D.M. Tilbury, MIMO Control of an Apache Web Server: Modeling and Controller Design, American Control Conference, 2002.

[16] Y. Diao, J.L. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano, Incorporating Cost of Control into the Design of a Load Balancing Controller, IEEE RTAS 2004.

[17] C. Dovrolis, D. Stiliadis, and P. Ramanathan, Proportional Differentiated Services: Delay Differentiation and Packet Scheduling, SIGCOMM, August 1999.

[18] P. Druschel and G. Banga, Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems, OSDI, 1996.

[19] L. Eggert and J. Heidemann, Application-Level Differentiated Services for Web Servers," World Wide Web Journal, 2(3), March 1999.

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext Transfer Protocol -- HTTP/1.1, IETF RFC 2616, June 1999.

[21] G.F. Franklin, J.D. Powell and A. Emami-Naeini, Feedback Control of Dynamic Systems, 1994.

[22] K. Jeffay, F.D. Smith, A. Moorthy, and J.H. Anderson, Proportional Share Scheduling of Operating System Services for Real-Time Applications, IEEE Real-Time Systems Symposium, December 1998.

[23] M. Karlsson, C. Karamanolis and X. Zhu, Triage: Performance Isolation and Differentiation for Storage Systems, IWQoS 2004.

[24] B. Li and K. Nahrstedt, A Control-based Middleware Framework for Quality of Service Adaptations, IEEE Journal of Selected Areas in Communication, Sept. 1999.

[25] C.L. Liu and J.W. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, Journal of ACM, 20(1): 46-61, 1973.

[26] C. Lu, T.F. Abdelzaher, J.A. Stankovic, and S.H. Son, A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers, IEEE Real-Time Technology and Applications Symposium, June 2001.

[27] C. Lu, G.A. Alvarez, and J. Wilkes, Aqueduct: Online Data Migration with Performance Guarantees, USENIX Conference on File and Storage Technologies, January 2002.

[28] C. Lu, J.A. Stankovic, G. Tao and S.H. Son, Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms, Real-Time Systems, 23(1/2), 2002.

[29] Y. Lu, T.F. Abdelzaher, C. Lu, and G. Tao, An Adaptive Control Framework for QoS Guarantees and Its Application to Differentiated Caching Services, IWQoS 2002.

[30] Y. Lu, T.F. Abdelzaher, C. Lu, L. Sha, and X. Liu, Feedback Control with Queueing-Theoretic Prediction for Relative Delay Guarantees in Web Servers, IEEE RTAS 2003.

[31] J. C. Mogul, The Case for Persistent-Connection HTTP, SIGCOMM, Cambridge, MA, 1995.

[32] V. Pai, P. Druschel and W. Zwaenepoel, Flash: An Efficient and Portable Web Server, USENIX Annual Technical Conference, June 1999.

[33] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus, Using Control Theory to Achieve Service Level Objectives in Performance Management, IFIP/IEEE International Symposium on Integrated Network Management, May 2001.

[34] S. Parekh, K. Rose, Y. Diao, V. Chang, J.L. Hellerstein, S. Lightstone, and M. Huras, Throttling Utilities in the IBM DB2 Universal Database Server, American Control Conference, 2004.

[35] L. Sha, X. Liu, Y. Lu, and T.F. Abdelzaher, Queueing Model Based Network Server Performance Control, IEEE Real-Time Systems Symposium, 2002.

[36] D.C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, A Feedback-driven Proportion Allocator for Real-Rate Scheduling, OSDI, Feb 1999.

[37] N. Vasiliou and H. Lutfiyya, Providing a Differentiated Quality of Service in a World Wide Web Server, Performance Evaluation Review, 28(2): 22-27.

[38] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, RSVP: A New Resource ReSerVation Protocol, IEEE Network, September 1993.