A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers*

Chenyang Lu Tarek F. Abdelzaher John A. Stankovic Sang H. Son
Department of Computer Science, University of Virginia
Charlottesville, VA 22903
e-mail: {chenyang, zaher, stankovic, son}@cs.virginia.edu

Abstract

This paper presents the design, implementation, and evaluation of an adaptive architecture to provide relative delay guarantees for different service classes on web servers under HTTP 1.1. The first contribution of this paper is the architecture based on a feedback control loop that enforces desired relative delays among classes via dynamic connection scheduling and process reallocation. The second contribution is our use of feedback control theory to design the feedback loop with proven performance guarantees. In contrast with ad hoc approaches that often rely on laborious tuning and design iterations, our control theory approach enables us to systematically design an adaptive web server with established analytical methods. The design methodology includes using system identification to establish a dynamic model, and using the Root Locus method to design a feedback controller to satisfy performance specifications of a web server. The adaptive architecture has been implemented by modifying an Apache web server. Experimental results demonstrate that our adaptive server achieves robust relative delay guarantees even when workload varies significantly. Properties of our adaptive web server include guaranteed stability, and satisfactory efficiency and accuracy in achieving the desired relative delay differentiation.

1. 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) motivate building QoS-aware web servers. Such servers customize their performance attributes depending on the class of the served requests such that more important requests receive better service. From the perspective of the requesting clients, the most visible service performance attribute is typically the server latency. Different requests may have different tolerances to latency. For example, one can argue that stock trading requests should be served more promptly than information requests. Similarly, interactive clients should be served more promptly than background software agents such web crawlers and prefetching proxies. Some businesses may also want to provide different service latency to different classes of customers depending on their monthly fees or importance. Hence, in this paper, we provide a solution to support delay differentiation in web servers.

Support for different classes of service on the web (with special emphasis on server delay differentiation) has been investigated in recent literature. In the simplest case, it is proposed to differentiate between two classes of clients; premium and basic. For example, the authors of [21] proposed and evaluated an architecture in which restrictions are imposed on the amount of server resources (such as threads or processes) available to basic clients. In [5][6] admission control and scheduling algorithms are used to provide premium clients with better service. In [10] a server architecture is proposed that maintains separate service queues for premium and basic clients, thus facilitating their differential treatment.

While the above differentiation approach usually offers better service to premium clients, it does not provide any *guarantees* on the service. Hence, we call these approaches the *best effort differentiation* model. In particular, the best effort differentiation model does not provide guarantees on the extent of the difference between premium and basic performance levels. This difference depends heavily on load conditions and may be difficult to quantify. In a situation where clients pay to receive better service, any ambiguity regarding the expected performance improvement may cause client concern, and is, therefore, perceived as a disadvantage. Compared with the best effort differentiation model, the *absolute guarantee* model and the *proportional differentiated service* model both provide stronger guarantees on service differentiation.

In the absolute guarantee model, a fixed maximum service delay (a soft deadline) for each class needs to be

* Supported in part by NSF grants CCR-9901706, CCR-0093144, CCR-0098269, and EIA-9900895, and contract IJRP-9803-6 from the Ministry of Information and Communication of Korea.

enforced. A disadvantage of the absolute guarantee model is that it is usually difficult to determine appropriate deadlines for web services. For example, the tolerable delay threshold of a web user may vary significantly depending on web page design, length of session, browsing purpose, and properties of the web browser [12]. Since system load can grow arbitrarily high in a web server, it is impossible to satisfy the absolute delay guarantees of all service classes under overload conditions. In the absolute guarantee model, deadlines that are too loose may not provide necessary service differentiation because the deadlines can be satisfied even when delays of different classes are the same. On the other hand, deadlines that are too tight can cause extremely long latency or service rejections in low priority classes in order to enforce high priority classes' (potentially unnecessarily) tight deadlines.

In the proportional differentiated service model introduced in [19], a fixed ratio between the delays seen by the different service classes can be enforced. This architecture provides a specification interface and an enforcement mechanism such that a desired "distance" between the performance levels of different classes can be specified and maintained. This service model is more precise in its performance differentiation semantics than the best effort differentiation model. The proportional differentiated service is also more flexible than absolute guarantees because it does not require fixed deadlines being assigned for each service class.

Depending on the nature of the overload condition, either the proportional differentiated service or the absolute guarantee may become more desirable. The proportional differentiated service may be less appropriate in severe overload conditions because even high priority clients may get extremely long delays. In nominal overload conditions, however, the proportional differentiated service may be more desirable than absolute guarantee because the proportional differentiated service can provide adequate and precise service differentiation without requiring artificial, fixed deadlines being assigned to each service class. Therefore, a *hybrid* guarantee might be desirable in some systems. For example, a hybrid policy can be that the server provides proportional differentiated service when the delay received by each class is within its tolerable threshold. When the delay received by a high priority class exceeds its threshold, the server automatically switches to the absolute guarantee model that enforces desired delays for high priority classes at the cost of violating desired delays of low priority classes. This policy can achieve the flexibility of the proportional differentiated service in nominal overload and bound the delay of high priority class in severe overload conditions.

In this paper, we present an architecture to support the proportional differentiated service model introduced in [19]. This architecture provides a specification interface and an enforcement mechanism such that a desired "distance" between the performance levels of different classes can be specified and maintained. This service model is more precise in its performance differentiation semantics than the best effort differentiation model. The proportional differentiated service is also more flexible than absolute guarantee because it does not require fixed deadlines being assigned for each service class. The absolute and hybrid delay guarantees are addressed in [26].

A key challenge in implementing proportional differentiated services in a web server is that resource allocation that achieves the desired degree of differentiation depends on load conditions that are unknown *a priori*. 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 [22], the most recent version of HTTP that has been adopted by most web servers and browsers. As we show in this paper, persistent connections introduced by HTTP 1.1 give rise to peculiar server bottlenecks that affect our choice of resource allocation mechanisms for delay differentiation. Our contributions can be summarized as follows:

- An adaptive architecture for achieving proportional differentiated service delay in web servers under HTTP 1.1
- Use of feedback control theory and methodology to design an adaptive connection scheduler with proven performance guarantees. The design methodology includes
  - Using system identification to model web servers for purposes of performance control
  - Specifying performance requirements of web servers using control-based metrics
  - Designing feedback controllers using the Root Locus method to satisfy the performance specs
- Implementing and evaluating the adaptive architecture on a modified Apache web server.

The rest of this paper is organized as follows. In Section 2, we briefly describe how web servers (in particular with HTTP 1.1 protocol) operate. In Section 3, we define the relative delay differentiation problem on web servers. The design of the adaptive server architecture to achieve relative delay guarantees is described in Section 4. In Section 5, 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 web server and experimental results are presented in Sections 6 and 7,

respectively. The related work is summarized in Section 8. We then conclude the paper in Section 9.

2. Background

The first step towards designing architectural components for service delay differentiation is to understand how web servers operate. Web server software usually adopts a multi-process or a multi-threaded model. Processes or threads can be either created on demand or maintained in a pre-existing pool that awaits incoming TCP connection requests to the server. In this paper, we assume a multi-process model with a pool of processes, which is the model of the Apache server, the most commonly used web server today.

In HTTP 1.0, each TCP connection carried a single web request. This resulted in an excessive number of concurrent TCP connections. To remedy these problems the current version of HTTP, called HTTP 1.1, reduces the number of concurrent TCP connections with a mechanism called *persistent connection* [22], which allows multiple web requests to reuse the same connection. An HTTP 1.1 client first sends a TCP connection request to a web server. The request is stored in the listen queue of the server's well-known port. Eventually, the request is dequeued and a TCP connection is established between the client and one of the server processes. The client can then send HTTP requests and receive responses over the established connection. The HTTP 1.1 protocol requires that an established TCP connection be kept alive after a request is served in anticipation of potential future requests. If no requests arrive on this connection within TIMEOUT seconds, the connection is closed and the process responsible for it is returned to the idle pool. Due to the increasing popularity of HTTP 1.1, we focus on the implications of persistent connections on server performance, and present a resource allocation mechanism for delay differentiation that specifically addresses the peculiarities of this protocol.

Persistent connections generate a new type of resource bottleneck on the server. Since a server process may be tied up with a persistent connection even after the request is served, the CPU(s) can be under-utilized. One way to alleviate this problem is to increase the number of server processes. However, too many processes can cause thrashing in virtual memory systems [16] thus degrading server performance considerably. In practice, an operating system specific limit is imposed on the number of concurrent server processes. This limit is often the bottleneck in servers implementing HTTP 1.1. Hence, while new connections suffer large queuing delays, requests arriving on existing connections are served almost immediately by their dedicated processes[1]. We verify this observation experimentally as described in Section 7.1. The observation is important as it affects our choice of delay metrics and QoS enforcement architecture. In particular, since CPU may not be the bottleneck resource, CPU scheduling/allocation is not an effective mechanism to provide service differentiation in web servers with HTTP 1.1. Instead, we develop a server process allocation mechanism (Section 4.1) to support service differentiation in such systems. Note that it is not the objective of our current research to change HTTP 1.1 semantics to improve CPU utilization.

3. The Relative Delay Guarantee

Let the *connection delay* denote the time interval between the arrival of a TCP connection request and the time the connection is accepted (dequeued) by a server process. Let the *processing delay* denote the time interval between the arrival of an HTTP request to the process responsible for the corresponding connection and time the server completes transferring the response.

Connection delay includes the queuing delay on the server's well known port. As explained in the previous section, such queuing delay may be significant even when CPU utilization on the server is not high. Processing delay of requests on already established connections, on the other hand, is much smaller. Hence, we focus on applying delay differentiation only to connection delays. Using connection delays as the delay metric of choice is also desirable for another reason. Besides being the dominant delay, it is also less of a function of client-side limitations than the processing delay. The processing delay is dominated by the time is takes to send the response to the client which depends on TCP throughput. If the client is slow, TCP flow control will reduce the response transfer rate accordingly. Since processing delay depends on client speed, it is not an appropriate metric of performance quality that is attributable to the server. Instead, our mechanism provides relative connection delay guarantees. A relative connection delay guarantee is defined as follows.

Every HTTP request belongs to a class k ($0 \le k < N$). A *desired relative delay* W_k is assigned to each class k. $\{W_k \mid 0 \le k < N\}$ establishes a *QoS contract* between the system and its users. The connection delay $C_k(m)$ of class k at the m^{th} sampling instant is defined as the average connection delay of all established connections of class k within the $((m-1)S, mS)$ sec, where S is a constant sampling period. A *relative connection delay*

1 Several new web servers such as [30] have a single-threaded and event driven architecture that has no limit on the number of connections that can be accepted besides limits on the number of open descriptors imposed by the OS. In this architecture, processing delay instead of the connection delay may dominate the server response time.

guarantee requires that $C_j(m)/C_l(m) = W_j(m)/W_l(m)$ for $\forall$ classes j and l ($j \neq l$). 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. We will simply use delay to refer to connection delay in the rest of the paper.

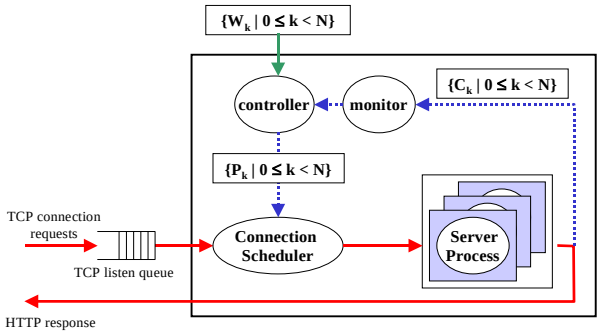


Figure 1: The Feedback-Control Architecture

4. The Feedback Control Architecture

In this section, we present an adaptive web server architecture (as illustrated in Figure 1) to achieve relative delay guarantees. A key feature of this architecture is a feedback control loop that maintains desired relative delays via dynamic reallocation of server process. The architecture is composed of a Connection scheduler, a Monitor, a Controller, and a fixed pool of server processes. We describe the design of the components in the following subsections.

4.1. Connection Scheduler

The Connection Scheduler serves as an actuator to control the relative delays of different classes. It listens to the well-known port and accepts every incoming TCP connection request. The Connection Scheduler uses an adaptive proportional share policy to allocate server processes to connections from different classes. Assuming the *process proportions* at sampling instant m of all classes are $\{P_k(m) \mid 0 \leq k < N\}$ and the total number of server processes is M, each class k should be allocated at most $(P_k(m)/\sum_i P_j(m))M$ server processes. Note that $P_k(m)$ represents class k's *relative share* (instead of the absolute number) of server processes. For each class k ($0 \leq k < N$), the Connection Scheduler maintains a (FIFO) connection queue Q_k and a process counter R_k. The connection queue Q_k holds connections of class k before they are allocated server processes. The counter $R_k(m)$ represents 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 to 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 $\{j \mid R_j < (P_j/\sum_l P_l)M\}$.

For the above scheduling algorithm, a key issue is how to decide the process proportions $\{P_k \mid 0 \leq k < N\}$ to achieve the desired relative delay among classes $\{W_k \mid 0 \leq k < N\}$. Note that a static mapping from the response time weight $\{W_k \mid 0 \leq k < N\}$ to $\{P_k \mid 0 \leq k < N\}$ based on system profiling cannot work well when the workloads are unpredictable and can vary at run time (see performance results in Section 7.3.1). This problem motivates the use of a feedback controller to dynamically adjust the process proportions $\{P_k \mid 0 \leq k < N\}$ to maintain desired relative delays.

Because process proportions can be dynamically changed by the Controller, a situation can occur when a class k's process proportion is increased, there are not enough available server processes in other classes to be reallocated to class k. Two different policies, preemptive vs. non-preemptive scheduling, can be supported in this case. In the preemptive scheduling policy, the Connection Scheduler immediately forces server processes to close other classes' connections to reclaim enough server processes for class k. In the non-preemptive scheduling model, the Connection Scheduler waits for server processes to voluntarily release connections of other classes before it allocates enough processes to class k. The advantage of the preemptive scheduling is that it is more responsive to the Controller's input and therefore can adapt faster to load variations, but it can also cause jittery response time in preempted classes because they may have to re-establish a connection with the server in the middle of loading a web page. Only the non-preemptive scheduling is currently implemented in our web server. The preemptive scheduling will be investigated in our future work.

4.2. Server Processes

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

4.3. Monitor

At each sampling instant m, the Monitor is invoked to compute the average connection delays $\{C_k(m) \mid 0 \leq k < N\}$ of all classes during the last sampling period. The sampled connection delays are used by the Controller to calculate new process proportions.

4.4. Controller

At each sampling instant m, the Controller compares the sampled connection delays $\{C_k(m) \mid 0 \le k < N\}$ with the desired relative delays $\{W_k \mid 0 \le k < N\}$, and computes new process proportions $\{P_k(m+1) \mid 0 \le k < N\}$, which are subsequently used by the Connection Scheduler to reallocate server processes during the following sampling period. The goal of the Controller is to maintain desired relative delays $\{W_k \mid 0 \le k < N\}$. We first design a Basic Controller to achieve desired relative delays for two classes in Section 4.4.1. In Section 4.4.2, we design a General Controller for N classes that is composed of N-1 Basic Controllers.

4.4.1. A Basic Controller for Two Classes

Before we describe the Basic Controller, we make the following definitions.

- For the purpose of controlling the relative delay between class k and k-1, the input (reference) to a Basic Controller $CTRL_k$ is the desired delay ratio between class k and k-1, i.e., W_k/W_{k-1}. The output of a web server at the m^{th} sampling instant is the sampled delay ratio between class k and k-1, i.e., $V_k(m) = C_k(m)/C_{k-1}(m)$. The difference between the reference and the output is the error $E_k(m)=W_k/W_{k-1} - V_k(m)$.
- At every sampling point m, the Basic Controller computes the current process ratio (control input) $U_k(m)=P_{k-1}(m)/P_k(m)$, i.e., the ratio between the number of processes allocated to class k-1 and k.

Intuitively, when $E_k(m) < 0$, the Controller should decrease the process ratio $U_k(m)$ to allocate more processes to class k relative to class k-1. The goal of the controller $CTRL_k$ is to reduce the error $E_k(m)$ to 0 and achieve the correct delay ratio between class k and k-1. The Basic Controller uses PI (Proportional-Integral) control [23] to compute the control input. A digital form of the PI control function is

$$U_k(m) = U_k(m-1) + g(E_k(m) - rE_k(m-1)) \qquad (1)$$

where g and r are design parameters called the *controller gain* and the *controller zero*, respectively. The performance of the web server significantly depends on the values of the controller parameters. An *ad hoc* approach to design the controller is to conduct laborious experiments on different values of the parameters. In our work, we apply control theory to tune the parameters analytically to guarantee the desired performance in the web server. The design method is presented in Section 5.

4.4.2. A General Controller

A General Controller for N classes is composed of N-1 Basic Controllers $\{CTRL_k \mid 1 \le k < N\}$. At every sampling instant m, the General Controller sets class (N-1)'s process proportion $P_{N-1}(m)=1$ (note that process allocations only depend on the relative values of proportions), and then runs a loop to compute all the process proportions $P_k(m)$ in the decreasing order of k (from N-2 to 0). In each iteration of the loop, a Basic Controller $CTRL_k$ is called to compute the process ratio $U_{k+1}(m)$ between class k and k+1, and the process proportion of class k $P_k(m) = P_{k+1}(m)U_k(m)$. This loop iterates until all the process proportions are computed[2].

In summary, we have presented a feedback control architecture to achieve relative delay guarantees on web servers. A key component in this architecture is the Controller that dynamically computes correct process proportions in face of workload variations. In the rest of the paper, we use the *closed-loop server* to refer to the adaptive web server with the Controller (Figure 1), while the *open-loop server* refers to a non-adaptive web server without the Controller.

5. Design of the Controller

In this Section, we apply a control theory framework [27] to design the Basic Controller $CTRL_k$ that guarantees the desired relative delay between two neighboring classes k-1 and k. We focus on the Basic Controller because the performance of the general Controller depends on multiple Basic Controllers. In Section 5.1, we specify the performance requirement of the Basic Controller. We then use system identification techniques to establish a dynamic model for the web server in Section 5.2. Based on the dynamic model, we use the Root Locus method to design a Basic Controller that satisfies the performance specifications.

5.1. Performance Specs

In [27], a set of performance metrics was presented to characterize the performance of adaptive real-time systems. Compared with traditional metrics that only describe steady state performance, the specs and metrics presented in [27] can characterize the dynamic performance of adaptive systems in both transient and steady state. In this paper, we use similar metrics to specify the performance requirements of the closed-loop server. The performance specs of the closed-loop server include in following.

- **Stability**: a (BIBO) stable system should have bounded output in response to bounded input. For a

[2] A different design of the General Controller may use multi-input-multi-output control to directly computes the process proportions based on the delays of all service classes. We will investigate this approach in our future work.

closed-loop server, stability requires the actual delay ratio should always be bounded at run-time. Stability is a necessary condition for achieving desired relative delay.

- **Settling time T_s** is the time it takes the output to converge to the vicinity of the reference and enters a steady state. For a closed-loop server, the settling time represents the efficiency of the Controller, i.e., how fast the server can converge to the desired relative delay. As an example, we assume that our web server requires the settling time $T_s < 300$ sec.
- **Steady state error V_s** is the difference between the reference input and average of output after the system enters a steady state. For a closed-loop server, the steady state error represents the accuracy of the Basic Controller in achieving the desired relative delays. As an example, we assume that our web server requires a steady state error $|V_s| < 0.1W_k/W_{k-1}$. Note that satisfying this steady state error requirement means that our web server can achieve the desired relative delays in steady state.

5.2. System Identification

A dynamic model describes the mathematical relationship between the input and the output of a system (usually with differential or difference equations). Modeling is important because it provides a basis for the analytical design of a controller. To design a Basic Controller $CTRL_k$ to achieve desired delay ratio between classes k and k-1, we need a dynamic model for the open-loop server. For the purpose of performance control, the *input* of the model is the process ratio $U_k(m) = P_{k-1}(m)/P_k(m)$. The *output* of the model is the delay ratio $V_k(m) = C_k(m)/C_{k-1}(m)$. However, unlike traditional control theory applications such as mechanical and electronic systems, it is difficult to directly describe a computing system such as a web server with differential or difference equations. To solve the modeling problem, we adopt a practical approach by applying system identification [8] to estimate the model of the web server. The open-loop server (including the Connection scheduler, the server processes, and the Monitor) is modeled as a difference equation with unknown parameters. We then stimulate the web server with pseudo-random digital white-noise inputs [29] and use a least squares estimator to estimate the model parameters. Our experimental results (Section 7.2) established that the open-loop server can be modeled as a second order difference equation with adequate accuracy for the purpose of control design. The architecture used for system identification is illustrated in Figure 2. We describe the components of the architecture in the following subsections.

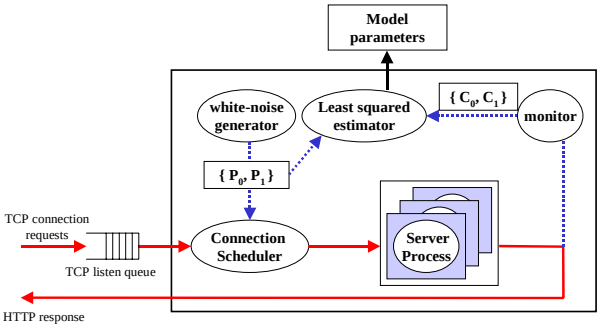


Figure 2: The System Identification Architecture

5.2.1. Model Structure

The web server is modeled as a difference equation with unknown parameters, i.e., an n^{th} order model can be described 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 is motivated by our observation that the output of an open-loop server depends on previous inputs and outputs (experimental results are not shown in this paper due to space limitations). Intuitively, the dynamics of a web server is due to the queuing of connections and the non-preemptive scheduling mechanism. For example, the connection delay may depend on the number of server processes allocated to its class in several previous sampling periods. For another example, after class k's process proportion is increased, the Connection Scheduler has to wait for connections of other classes to voluntarily release server processes to reclaim enough processes to class k.

5.2.2. White Noise Input

To stimulate the dynamics of the open-loop server, we use a standard pseudo-random digital white noise generator to randomly switch the process ratios between two different levels. White noise input has been commonly used for system identification [8]. The standard white noise algorithm is not presented due to space limitations and can be found in [29].

5.2.3. Least Squares Estimator

The least squares estimator is the key component of the system identification architecture. 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 [8]. 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 pre-

vious inputs $U(m-j)$ ($1 \le j \le n$). The measured output $V(m)$ is fit to the model as described in Equation (2). Define the vector $q(m) = (V(m-1) \ldots V(m-n)\ U(m-1) \ldots U(m-n))^T$, and the vector $\theta(m) = (a_1(m) \ldots a_n(m)\ b_1(m) \ldots b_n(m))^T$, i.e., the estimations 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 [8]:

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

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

$$R(m) = R(m-1)(I - q(m) \gamma(m) q(m)^T R(m-1)) \quad (5)$$

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

Our experimental results (Section 7.2) established that the web server can be modeled as a second order difference equation,

$$V(m) = a_1 V(m\text{-}1) + a_2 V(m\text{-}2) + b_1 U(m\text{-}1) + b_2 U(m\text{-}2) \quad (6)$$

where the estimated model parameters are (a_1, a_2, b_1, b_2) = (0.74, -0.37, 0.95, -0.12).

5.3. Root-Locus Design

Given a model described by Equation (6), we can apply control theory methods such as the Root Locus [23] to design the Basic Controller. The open loop server model in Equation (6) can be converted to a transfer function $G(z)$ in z-domain (Equation (7)). The transfer function of the PI controller (Equation (1)) in the z-domain is Equation (8). Given the open-loop model and the controller model, the transfer function of the closed loop server is Equation (9).

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

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

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

According to control theory, the performance of a system depends on the poles of its transfer function. The Root Locus is a graphical technique that plots the traces of poles of a closed-loop system on the z-plane (or s-plane) as its controller parameters change. We used the Root Locus tool of MATLAB to tune the controller gain g and the controller zero r so that the performance specs can be satisfied. Due to space limitations, we only summarize results of the design in this paper. The details of the design process can be found in control textbooks such as [23]. For the closed-loop server modeled as Equation (9), the traces of its poles as the controller gain increases are illustrated on the z-plane in Figure 3. We placed the closed-loop poles $p_0 = 0.70$ and $p_{1,2} = 0.38 \pm 0.62i$ (see Figure 3) by setting the controller gain $g = 0.3$ and the controller zero $r = 0.05$. Based on control theory [23], our closed-loop server has the following properties.

- **Stability**: The closed-loop server guarantees stability because all the closed-loop poles are in the unit circle, i.e., $|p_j| < 1$ ($0 \le j \le 2$).
- **Settling time**: According to control theory, decreasing the radius (the distance to the origin in the z-plane) of the closed-loop poles usually results in shorter settling time. As a trade-off among the radius of the three closed-loop poles, we use the Root Locus tool to place the closed-loop poles 0.70 and $0.38 \pm 0.62i$ to achieve a settling time of 270 sec, lower than the required settling time (300 sec) in Section 5.1.
- **Steady state error**: The closed-loop server achieves zero steady state error. This result can be easily proved using the Final Value Theorem in digital control theory [23]. This means that the closed-loop server can guarantee the desired relative delays in steady state.

In summary, using feedback control theory including system identification and the Root Locus design, we systematically designed the Basic Controller that *analytically* guarantees the performance specs. This result shows the strength of the control-theory-based framework for adaptive computer systems. A remaining question is whether the General Controller composed of multiple Basic Controllers can still satisfy the performance specifications because the concurrently running Basic Controllers may disturb each other's performance. The rigorous analysis of the General Controller is part of future work.

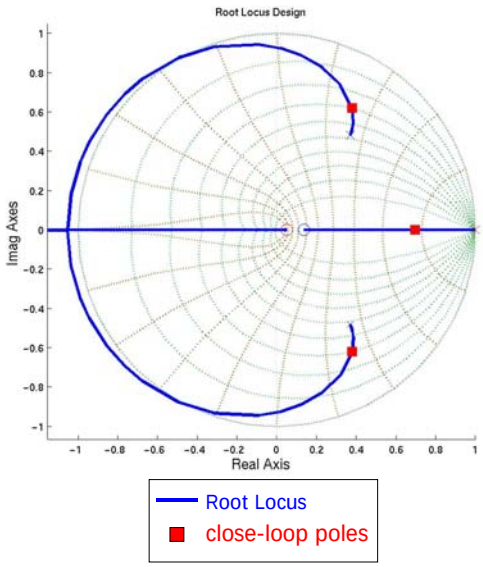


Figure 3 The Root Locus of the web server model

6. Implementation

We modified the source code of Apache 1.3.9 [7] and added a new library that implemented a Connection Manager (including the Connection Scheduler, the Monitor and the Controller). 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. Each connection request is classified based on its sender's IP address[3] and scheduled by a Connection Scheduler function. The Connection Scheduler dispatches a connection by sending its descriptor to a free server process through the corresponding 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 of the connection. 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 all incoming connection request in a tight loop.
- A Monitor and a Controller function are invoked periodically at every sampling instance. For each invocation, the Monitor computes the average delay for each class. This information is then passed to the Controller function, which implements the control algorithm to compute new process proportions.
- We modified the code of the server processes so that they accept connection descriptors from UNIX domain sockets (instead of common TCP listen socket as in Apache). When a server process closes a connection, it notifies the Connection Manager of its new status by sending a byte of data to the Connection Manager through the UNIX domain socket.

The server can be configured to a closed-loop/open-loop server by turning on/off the Controller. An open-loop server can be configured for system identification or performance evaluation.

7. Experimentation

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

- **Client:** We used SURGE [11] to generate realistic web workloads in our experiments. SURGE uses a number of *users* 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 500 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 (the delays in our experiments were therefore larger than normal servers). 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 sampling period S was set to 30 sec in all the experiments. The connection TIMEOUT of HTTP 1.1 was set to 15 sec.

In Section 7.1, we present experimental results that compare connection delays with response times of a web server. The system identification experiments are

3 Other criteria for connection classification include HTTP cookies, Browser plug-ins, URL request type or filename path, and destination IP address of virtual servers [10].

presented in Section 7.2. We present the evaluation of the closed-loop server in Section 7.3.

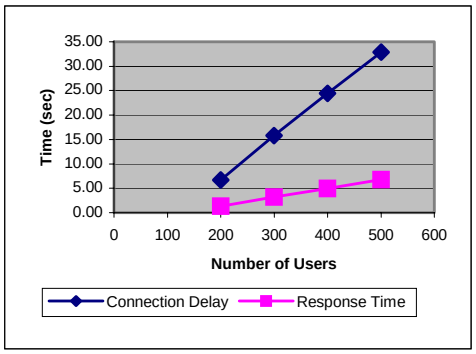


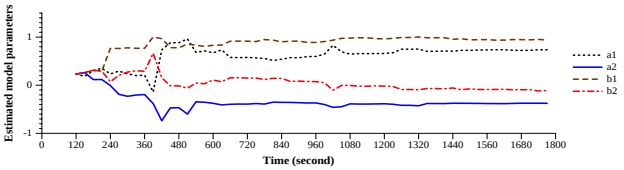
Figure 4: Connection delay and response time

7.1. Connection Delays and Response Times

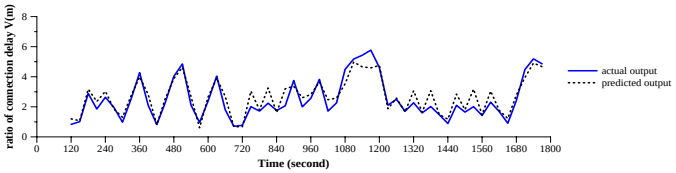
In the first set of experiments, we compare the average connection delay and the average response time (per HTTP request) of an open-loop server to justify the use of connection delay as the metric for delay differentiation in web servers with HTTP 1.1. All connections are treated as being in a same class and all server processes are allocated to the class. The average response time is computed based on the response times of all HTTP requests recorded in the logs of the SURGE clients, and the average connection delay is collected by the monitor on the modified Apache server. Every point in Figure 4 refers to the average connection delay or average response time in four 10-minute runs with a same number of users. The 90% confidence intervals are within 0.58 sec to all the presented average connection delays, and within 0.21 sec to all the presented average response times. The connection delay is significantly higher and increases faster than the response time as the number of users increases. For example, when the number of users is 400, the average connection delay is 24.47 sec, 4.9 times the average response time 4.98 sec. Two types of requests contribute to the average response time: the response time (including the connection delay and the processing delay) of the first request of each TCP connection and the response time (including only the processing time) of each subsequent request on the connection[4]. The difference between connection delay and response time demonstrates that the processing delay is on average significantly shorter than the connection delay. We also run similar experiments with 256 server processes (the maximum number allowed by the original Apache server on Linux). With 256 server processes, the ratio between the connection delay and the response time is similar to Figure 4. For example, the connection delay is 5.3 times the response time when there are 400 users. This result justifies our decision to use connection delay as the metric for delay differentiation in web servers with HTTP 1.1.

7.2. System Identification

We now present the results of system identification to establish a dynamic model for the open-loop server. Four client machines were divided into two classes 0 and 1, and each class had 200 users. The process ratio $P_0(m)/P_1(m)$ was initialized to 1. At each sampling instant, the white noise randomly set the process ratio to 3 or 1. The sampled relative delay $C_1(m)/C_0(m)$ was fed to the least squares estimator to estimate model parameters in Equation (2). Figure 5(a) shows that the estimated parameters of a second order model (Equation (6)) at successive sampling instants in a 30 minute run. The estimator and the white noise generator were turned on two minutes after SURGE started in order to avoid its start-up phase. We can see that the estimations of the parameters (a_1, a_1, b_1, b_2) converged to (0.74, -0.37, 0.95, -0.12). Substituting the estimations into Equation (6), we established an estimated second-order model for the open-loop server. To verify the accuracy of the model, we re-ran the experiment with a different white noise input (i.e., with a different random seed) to the open-loop server and compared the actual delay ratio and that predicted by the estimated model. The result is illustrated in Figure 5(b). We can see that prediction of the estimated model is consistent with the actual relative delay all through the 30 min run. This result shows that the estimated second order model is adequate for control design. Our other experiments (not shown due to space limitations of this paper) also showed that an estimated first order model had larger prediction error than the second order model, while an estimated third order model did not improve the modeling accuracy.



(a) Estimated model parameters



(b) Acuracy of estimated model

Figure 5: System identification results

[4] The network transmission delays are negligible because the experiments are conducted on a 100 MB Ethernet.

7.3. Evaluation of the Adaptive Web Server

In this section, we present evaluation results for our adaptive web server. We first present the evaluation results of a server with two classes in Sections 7.3.1. The results for a server with three classes are presented in Section 7.3.2.

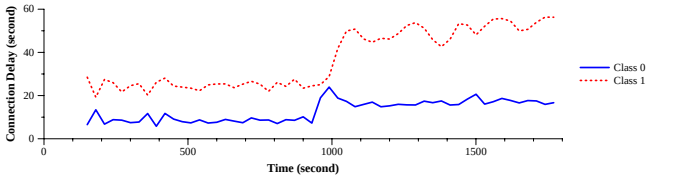
7.3.1. Evaluation of a Server with Two Classes

To evaluate a server with two classes, we set up the experiments as follows.

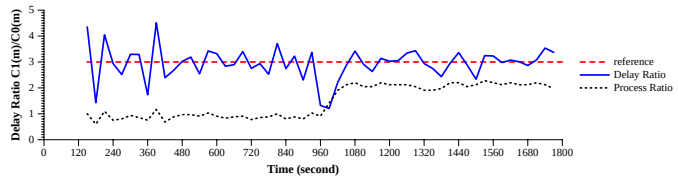
- **Workload**: Four client machines were evenly divided into two classes. Each client machine had 100 users. In the first half of each run, only one client machine from class 0 and two client machines from class 1 (100 users from class 0 and 200 users from class 1) generated HTTP requests to the server. The second machine from class 0 started generating HTTP requests 870 sec later than the other three machines. Therefore, the user population changed to 200 from class 0 and 200 from class 1 in the latter half of each run.
- **Closed-loop server**: The reference input (the desired delay ratio between class 1 and 0) to the Controller was W_1/W_0 = 3. The relative process proportion $P_0(m)/P_1(m)$ was initialized to 1 in the beginning of the experiments. To avoid the starting phase of SURGE, the Controller was turned on 150 sec after SURGE started. The sampled absolute connection delays and the delay ratio between the two classes are illustrated in Figure 6(a) and Figure 6(b), respectively.
- **Open-loop server**: An open-loop server was also tested as a baseline. The open-loop server was fine-tuned to have a "correct" process allocation based on profiling experiments using the original workload (100 class 0 users and 200 class 1 users). The results of the open-loop server are illustrated in (c)(d).

We first look at the first half of the experiment on the closed-loop server (Figure 6(a) and Figure 6(b)). When the Controller was turned on at 150 sec, the delay ratio $C_1(m)/C_0(m)$ = 28.5/6.5 = 4.4 due to incorrect process allocation. The Controller dynamically reallocated processes and changed the relative delay to the vicinity of the reference W_1/W_0 = 3. The relative delay stayed close (within 10%) to the reference at most sampling instants after it converged. This demonstrates that the closed-loop server can guarantee the desired relative delay. In terms of control metrics, the closed-loop server maintained stability because its relative delay was clearly bounded all through the run. Due to the noise of the server caused by the random workload, it is impossible to quantify the actual settling time and steady state error. However, we can see that the server rendered satisfactory efficiency and accuracy in achieving the desired relative delays.

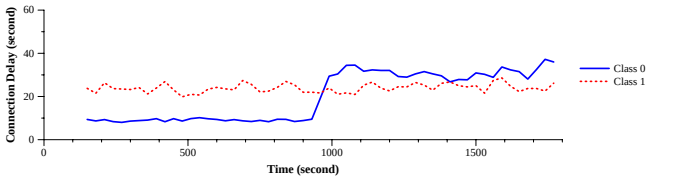
Compared with an open-loop server, a key advantage of a closed-loop server is that it can maintain robust relative delay guarantees when workload varies. Robust performance guarantees are especially important in web servers, which are often face with unpredictable and bursty workloads [18]. The robustness of our closed-loop server is demonstrated by its response to the load variation starting at 870 sec (Figure 6(a) and Figure 6(b)). Because the number of users of class 0 suddenly increased from 100 to 200, the delay ratio dropped from 3.2 (at 870 sec) to 1.2 (at 900 sec) - far below the reference W_1/W_0 = 3. The Controller reacted to load variation by allocating more processes to class 0 while deallocating processes from class 1. By time 1140 sec, the relative delay successfully re-converged to 2.9. In comparison, while the open-loop server achieved satisfactory relative delays when the workload was similar to its expectation (from 150 sec to 900 sec), it failed to maintain the desired relative delay after the workload changed (see Figure 6(a) and Figure 6(d)). Instead, connections from class 0 even consistently had longer delays than connections from class 1 after the workload changed, i.e., the open-loop server failed to maintain the desired relative delays after workload variations occurred.



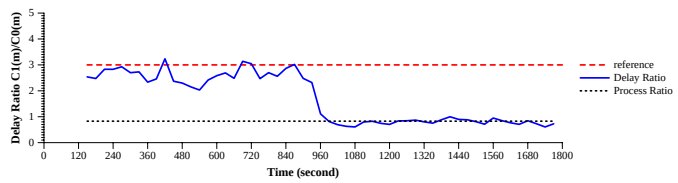
(a) Close-loop: Connection Delays C(0) and C(1)



(b) Close-loop: Delay Ratio (C1(m)/C0(m)) and Process Ratio (P0(m)/P1(m))



(c) Open-loop: Connection Delays C(0) and C(1)



(d) Open-loop: Delay Ratio (C1(m)/C0(m)) and Process Ratio (P0(m)/P1(m))

Figure 6: Performance results of Two classes

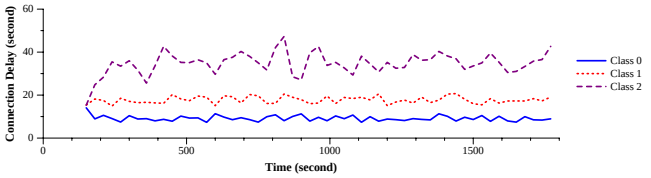


Figure 7: Performance Results of Three Classes

7.3.2. Evaluation of a Server with Three Classes

In the next experiment, we evaluate the performance of a closed-loop server with three classes. Each class had a client machine with 100 users. The Controller was turned on at 150 sec. The desired relative delay was (W_0, W_1, W_2) = (1, 2, 4). The process proportions were initialized to (P_0, P_1, P_2) = (1, 1, 1). From Figure 7, we can see that the connection delay began at (C_0, C_1, C_2) = (14.6, 17.3, 17.5) (equivalent to (1, 1.2, 1.2)), and then changed to (C_0, C_1, C_2) = (9.3, 16.2, 33.9) (equivalent to (1, 1.7, 3.6)) 240 sec after the Controller was turned on. The relative connection delay remained bounded and close to the desired relative delay in steady state. This experiment demonstrates that the general Controller composed of multiple Basic Controllers can provide desired relative delays for more than two classes.

In summary, our evaluation results demonstrate that the closed-loop server can achieve robust relative delay guarantees even when workload significantly varied. Properties of our adaptive web server include guaranteed stability, satisfactory efficiency and accuracy in achieving desired relative delay guarantees.

8. Related Work

Support for best-effort delay differentiation on web servers has been investigated in the recent literature [3][5][6][10][21]. Their work did not provide guarantees on the extent of the difference among service classes. Supporting proportional differentiated services in network routers has been investigated in [19][25]. Their work did not address end systems.

There have been several results that applied feedback control theory or principles to the design of computing systems. For example, several papers [2][9][14][15][20][34][36] presented adaptive CPU scheduling techniques to improve digital control system performance. These techniques are tailored to the specific characteristics of digital control systems. Several other papers [4][13][31][32][35] presented adaptive QoS management architectures for computing systems such as multimedia and communication systems. These solutions are mostly concerned with absolute metrics such as deadline miss-ratio and CPU/bandwidth utilizations rather than relative delays.

In [1], a least squares estimator was used for automatic profiling of resource usage parameters of a web server, but this work is not concerned with establishing a dynamic model for the server. We proposed a control-theory-based framework for adaptive real-time systems to guarantee low deadline miss-ratio in unpredictable environments in [27]. This paper extends the framework to web servers for relative service delay guarantees.

9. Conclusions and Future Work

The first contribution of this paper is a novel architecture to achieve relative delay guarantees for different service classes on web servers under HTTP 1.1. Our architecture features a feedback control loop that enforces desired relative delays among classes via dynamic connection scheduling and process reallocation. The second contribution of this paper is the use of feedback control theory to design the feedback loop with proven performance guarantees. In contrast to *ad hoc* approaches that rely on laborious design/tuning/testing iterations, our control theory approach enables us to systematically design an adaptive web server with established analytical methods. The design methodology includes using system identification to establish a dynamic model for a web server, and using the Root Locus method to design a feedback controller to satisfy performance specs. The adaptive architecture has been implemented by modifying an Apache web server. Experimental results demonstrate that our adaptive server achieves robust relative delay guarantees in face of significant workload variations. Our adaptive web server also guarantees stability and satisfactory efficiency and accuracy in achieving relative delay differentiation. In the future, we will extend our solutions to web server farms. We are also interested in investigating an adaptive control architecture that adjusts the Controllers based on on-line system identification. Such adaptive control scheme may further improve the robustness of the web servers in face of severe run-time variations.

Acknowledgements

The authors would like to thank Gang Tao, John Regehr, Jorg Liebeherr, and anonymous referees for their valuable suggestions to improve this paper.

10. Reference

[1] T. F. Abdelzaher, "An Automated Profiling Subsystem for QoS-Aware Services," *IEEE Real-Time Technology and Applications Symposium*, Washington D.C., June 2000.

[2] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin, "QoS Negotiation in Real-Time Systems and its Application to Automatic Flight Control," *IEEE Real-Time Technology and Applications Symposium*, June 1997.

[3] T. F. Abdelzaher and N. Bhatti, “Web Server QoS Management by Adaptive Content Delivery,” *International Workshop on Quality of Service*, 1999.

[4] T. F. Abdelzaher and K. G. Shin, "End-host Architecture for QoS-Adaptive Communication," *IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[5] T. F. Abdelzaher and K. G. Shin, “QoS Provisioning with qContracts in Web and Multimedia Servers,” *IEEE Real-Time Systems Symposium*, Phoenix, AZ, Dec 1999.

[6] J. Almedia, M. Dabu, A. Manikntty, and P. Cao, “Providing Differentiated Levels of Service in Web Content Hosting,” *First Workshop on Internet Server Performance*, Madison, WI, June 1998.

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

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

[9] G. Beccari, et. al., “Rate Modulation of Soft Real-Time Tasks in Autonomous Robot Control Systems,” *EuroMicro Conference on Real-Time Systems*, June 1999.

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

[11] P. Barford and M. E. Crovella, “Generating Representative Web Workloads for Network and Server Performance Evaluation,” *ACM SIGMETRICS '98*, Madison WI, 1998.

[12] 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*. Hague, Netherland, April 2000.

[13] S. Brandt and G. Nutt, “A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage,” *IEEE Real-Time Systems Symposium*, Dec 1998.

[14] G. Buttazzo, G. Lipari, and L. Abeni, "Elastic Task Model for Adaptive Rate Control", *IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec 1998.

[15] M. Caccamo, G. Buttazzo, and L. Sha, “Capacity Sharing for Overrun Control,” *IEEE Real-Time Systems Symposium*, Orlando, FL, Dec 2000.

[16] Carr, R., *Virtual Memory Management*, Ann Arbor, MI: UMI Research Press, 1984.

[17] S. Cen, "A Software Feedback Toolkit and its Application In Adaptive Multimedia Systems," *Ph.D. Thesis*, Oregon Graduate Institute, October 1997.

[18] 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, Dec 1997.

[19] C. Dovrolis, D. Stiliadis, and P. Ramanathan, “Proportional differentiated Services: Delay Differentiation and Packet Scheduling,” *SIGCOMM’99*, Cambridge, MA, August 1999.

[20] J. Eker: "Flexible Embedded Control Systems-Design and Implementation." PhD-thesis, Lund Institute of Technology, Dec 1999.

[21] L. Eggert and J. Heidemann, “Application-Level Differentiated Services for Web Servers,” *World Wide Web Journal*, Vol 2, No 3, March 1999, pp. 133-142.

[22] 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.

[23] G. F. Franklin, J. D. Powell and M. Workman, *Digital Control of Dynamic Systems (3rd Ed.)*, Addison-Wesley, 1997.

[24] B. Li and K. Nahrstedt, “A Control-based Middleware Framework for Quality of Service Adaptations,” *IEEE Journal of Selected Areas in Communication, Special Issue on Service Enabling Platforms*, 17(9), Sept. 1999.

[25] J. Liebeherr and N. Christin, “Buffer Management and Scheduling for Enhanced Differentiated Services,” *University of Virginia Tech. Report CS-2000-24*, August 2000.

[26] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, “A Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers,” *University of Virginia Tech Report CS-2001-06*, January 2001.

[27] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son and M. Marley, “Performance Specifications and Metrics for Adaptive Real-Time Systems,” *IEEE Real-Time Systems Symposium*, Orlando, FL, Dec 2000.

[28] C. Lu, J. A. Stankovic, G. Tao and S. H. Son, “Design and Evaluation of a Feedback Control EDF Scheduling Algorithm,” *IEEE Real-Time Systems Symposium*, Phoenix, AZ, Dec 1999.

[29] S. K. Park and K. W. Miller, “Random Number Generators: Good Ones Are Hard to Find”, *Communications of ACM*, vol. 21, no. 10, Oct. 1988.

[30] V. Pai, P. Druschel and W. Zwaenepoel, “Flash: An Efficient and Portable Web Server,” *USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[31] D. Rosu, K. Schwan, and S. Yalamanchili, “FARA–a Framework for Adaptive Resource Allocation in Complex Real-Time Systems,” *IEEE Real-Time Technology and Applications Symposium*, June 1998.

[32] D. Rosu, K. Schwan, S. Yalamanchili and R. Jha, "On Adaptive Resource Allocation for Complex Real-Time Applications," *IEEE Real-Time Systems Symposium*, Dec 1997.

[33] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao, "The Case for Feedback Control Real-Time Scheduling," *EuroMicro Conference on Real-Time Systems*, York, UK, June 1999.

[34] K. G. Shin and C. L. Meissner, “Adaptation and Graceful Degradation of Control System Performance by Task Reallocation and Period Adjustment,” *EuroMicro Conference on Real-Time Systems*, June 1999.

[35] D. C. Steere, et. al., "A Feedback-driven Proportion Allocator for Real-Rate Scheduling," *Symposium on Operating Systems Design and Implementation*, Feb 1999.

[36] L. R. Welch, B. Shirazi and B. Ravindran, “Adaptive Resource Management for Scalable, Dependable Real-time Systems: Middleware Services and Applications to Shipboard Computing Systems,” *IEEE Real-time Technology and Applications Symposium*, June 1998.