A Feedback Control Architecture and Design Methodology for Service Delay Guarantees 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 and implementation of an adaptive architecture to provide relative, absolute and hybrid service delay guarantees for different service classes on web servers under HTTP 1.1. The first contribution of this paper is the architecture based on feedback control loops that enforce delay guarantees for 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 dynamic models for a web server, and using the Root Locus method to design feedback controllers to satisfy performance specifications. The adaptive architecture has been implemented by modifying an Apache web server. Experimental results demonstrate that our adaptive server provides robust delay guarantees even when workload varies significantly. Properties of our adaptive web server also include guaranteed stability, and satisfactory efficiency and accuracy in achieving desired delay or 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) 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 delays. 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 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). 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 that differentiation should be made between two classes of clients; premium and basic. For example, the authors of [23] proposed and evaluated an architecture in which restrictions are imposed on the amount of server resources (such as threads or processes) which are available to basic clients. In [5][6] admission control and scheduling algorithms are used to provide premium clients with better service. In [11] a server

[*] Supported in part by NSF grants CCR-9901706 and EIA-9900895, and contract IJRP-9803-6 from the Ministry of Information and Communication of Korea. This paper is an extension to an earlier paper submitted to IEEE Real-Time Technology and Applications Symposium [33]. This paper is submitted to this special issue as an extension to [33] under the permission of the guest editors.

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 this approach 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 *proportional differentiated service* and the *absolute guarantee model* both provide stronger guarantees in service differentiation.

In the absolute guarantee model, a fixed maximum service delay (i.e., a soft deadline) for each class needs to be 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 [14]. 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. The absolute delay guarantee requires that all classes receive satisfactory delay if the server is not overloaded; otherwise desired delays are violated in the predefined priority order, i.e., low priority classes always suffer guarantee violation earlier than high priority classes[1]. 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 for low priority classes in order to enforce high priority classes' (unnecessary) tight deadlines.

In the proportional differentiated service model introduced in [21], 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 guarantee 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 is 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 a web server architecture to support delay guarantees including the absolute guarantee, proportional differentiated service, and the hybrid guarantee described above. A key challenge in guaranteeing service delays in a web server is that resource allocation that achieves the desired delay or

[1] Another scheme to implement absolute guarantee is to apply admission control on incoming requests during overload conditions. However, from the perspective of web clients, request denial by admission control is no better than service failure due to overload.

delay 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 [26], the most recent version of HTTP that has been adopted at present 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. Hence, our contributions can be summarized as follows:

- An adaptive architecture for achieving relative and absolute service delay guarantees 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, and
  - Designing feedback controllers using the Root Locus method to satisfy the performance specifications.
- Implementing and evaluating the adaptive architecture on a modified Apache web server that achieves robust performance guarantees even when the workload varies considerably.

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 formally define the semantics of delay differentiation guarantees on web servers. The design of the adaptive server architecture to the 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. The latter design alternative reduces service overhead by avoiding dynamic process creation and termination - a very costly operation in an operating system such as UNIX. 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 [25].

In HTTP 1.0, each TCP connection carried a single web request. This resulted in an excessive number of concurrent TCP connections. To remedy this problem the current version of HTTP, called HTTP 1.1, reduces the number of concurrent TCP connections with a mechanism called *persistent connections* [15], 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 responsibl

conditions. The absolute delay guarantee requires that all classes receive satisfactory delay if the server is not overloaded; otherwise desired delays are violated in the predefined priority order, i.e., low priority classes always suffer guarantee violation earlier than high priority classes.

Based on the relative and absolute delay guarantees, different hybrid guarantees can be composed for the specific requirements of the application. For example, the hybrid guarantee described in Section 1 can be formulated as follows.

- **A Hybrid Delay Guarantee**: Each class k is assigned a value W_k that represents both its desired delay and its desired relative delay. The hybrid guarantee $\{W_k \mid 0 \leq k < N\}$ provides the relative delay guarantees if the desired absolute delay of every class is satisfied. When the server is severely overloaded and desired delays cannot be provided to all classes, the hybrid guarantee provides absolute delay guarantees to high priority classes at the cost of violating the delays of low priority classes. This hybrid guarantee provides the flexibility of the proportional differentiated service in nominal overload while bounds the delay of high priority classes in severe overload conditions.

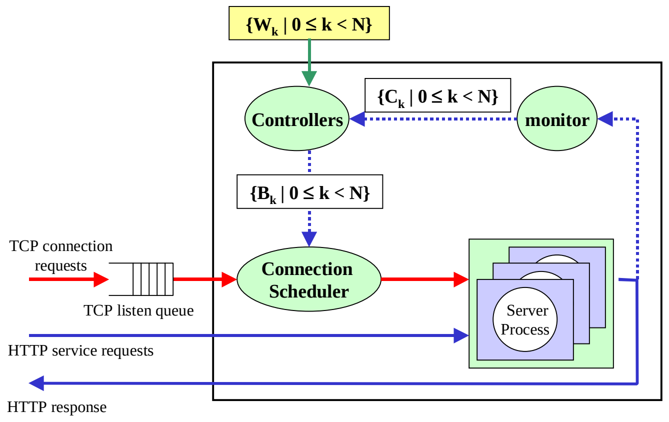


Figure 1 The Feedback-Control Architecture for Delay Guarantees

4. 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 relative/absolute 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 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[3]. At every sampling instant m, every class k $(0 \leq k < N)$ is assigned a *process budget*, $B_k(m)$, i.e., class k should be allocated at most $B_k(m)$ server processes in the m^{th} sampling period. For a system with absolute delay guarantees (Section 4.4.1)), the total budgets of all classes can exceed the total number of server

[3] Note that the Connection Scheduler uses process allocation instead of CPU allocation as a mechanism to control the delays of different classes. This is because processes may hold idle (persistent) connections and therefore CPU is not necessarily the bottleneck resource under HTTP 1.1 protocols (as discussed in Section 2).

processes in overload, which is a condition called *control saturation*. In this case, the process budgets are satisfied in the priority order until every process has been allocated to a class. This policy means that the process budgets of high priority classes are always satisfied before those of low priority classes, and thus the correct order of guarantee violations can be achieved. For a server with relative delay guarantee, our Relative Delay Controllers always guarantee that the total budget equals the total number of processes (Section 4.4.2). For each class k, 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 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 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 $\{j| R_j < B_j(m)\}$.

For the above scheduling algorithm, a key issue is how to decide the process budgets $\{B_k \mid 0 \le k < N\}$ 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 delay $\{W_k \mid 0 \le k < N\}$ to the process budget $\{B_k \mid 0 \le k < N\}$ (e.g., based on system profiling) cannot work well when the workloads are unpredictable and vary at run time (see performance results in Section 7.3.1). This problem motivates the use of feedback controllers to dynamically adjust the process budgets $\{B_k \mid 0 \le k < N\}$ to maintain desired delays.

Because the Controller can dynamically change the process budgets, a situation can occur when a class k's new process budget $B_k(m)$ (after the adjustment in saturation conditions described above) exceeds the total number of free server processes and processes already allocated to class k. Such class k 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 connections of *over-budget* classes whose new process budgets are less than the number of processes currently allocated to them. 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. The advantage of the preemptive model is that it is more responsive to the Controller's input and load variations, but it can 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. Only the non-preemptive model is currently implemented in our web server. The preemptive model will be investigated in our future work.

4.2. Server Processes

The second component of the architecture (Figure 1) 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

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

4.4. 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 $\{C_k(m) \mid 0 \le k < N\}$ with the desired relative or absolute delays $\{W_k \mid 0 \le k < N\}$, and computes new process budgets $\{B_k(m) \mid 0 \le k < N\}$[4], which are used by the Connection Scheduler to reallocate server processes during the following sampling

[4] It is the exact algorithm for this computation that control theory enables us to derive as described in the remainder of this section and Section 5.

period. We first describe the Absolute Delay Controllers and the Relative Delay Controllers in Sections 4.4.1 and 4.4.2, respectively. The Hybrid Delay Controllers based the Absolute and Relative Delay Controllers are described in Section 4.4.3.

4.4.1. The 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 in Table 1.

Reference VS_k	The reference of an Absolute Delay Controller CA_k is the desired delay of class *k*, i.e., $VS_k = W_k$.
Output $V_k(m)$	From the Absolute Delay Controller CA_k's perspective, the system output $V_k(m)$ at the sampling instant *m* is the sampled delay of class *k*, i.e., $V_k(m) = C_k(m)$.
Error $E_k(m)$	The difference between the reference and the output, i.e., $E_k(m) = VS_k – V_k(m)$.
Control input $U_k(m)$	At every sampling instant *m*, the Absolute Delay Controller CA_k computes the *control input* $U_k(m)$, i.e., the *process budget* $B_{k-1}(m)$ of class *k*.

Table 1: Variables and Parameters of the 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 achieve the desired delay for class *k*. Intuitively, when $E_k(m) = VS_k – V_k(m) < 0$, the Controller should increase the process budget $U_k(m) = B_k(m)$ to allocate more processes to class *k*. At every sampling instant *m*, the Absolute Delay Controller calls PI (Proportional-Integral) control [23] to compute the control input. A digital form of PI control function is

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

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 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 and tuning methodology is presented in Section 5.

For a system with *N* service classes, the Absolute Delay Guarantee is enforced by *N* Absolute Delay Controllers CA_k ($0 \le k < N$). At each sampling instant *m*, each Controller CA_k computes the process budget of class *k*. Note that in overload conditions, the process budgets (especially those of low priority classes) computed by the Absolute Delay Controllers may not be feasible if the sum of the computed process budgets of all classes exceeds the total number of server processes *M*, i.e., $\sum_j P_k(m) > M$. This is a situation called *control saturation*. Because low priority classes should suffer guarantee violation in overload conditions, the system always satisfy the computed process budgets in the decreasing order of priorities until every server process has been allocated to a class[5].

4.4.2. The Relative Delay Controllers

The relative delay of every two adjacent classes *k* and *k*-1 is controlled by a separate Relative Delay Controller CR_k. Each Relative Delay Controller CR_k, has following key parameters and variables. For simplicity of discussion, we use the same notations for the corresponding parameters and variables of the Absolute Delay Controller and the Relative Delay Controllers.

[5] To avoid complete starvation of low priority classes, the system may reserve a minimum number of server processes to each service class.

Reference VS_k	The reference of the Relative Delay Controller CR_k is the desired delay ratio between class *k* and *k*-1, i.e., $VS_k = W_k/W_{k-1}$.
Output $V_k(m)$	From the perspective of the Relative Delay Controller CR_k, the system output is the sampled delay ratio between class *k* and *k*-1, i.e., $V_k(m) = C_k(m) / C_{k-1}(m)$.
Error $E_k(m)$	The difference between the reference and the output, $E_k(m) = VS_k – V_k(m)$.
Control input $U_k(m)$	At every sampling instant *m*, CR_k computes the *control input* $U_k(m)$ defined as the ratio (called the *process ratio*) between the number of processes to be allocated to class *k*-1 and *k*, $U_k(m) = B_{k-1}(m) / B_k(m)$.

Table 2: Variables and Parameters of the Relative Delay Controller CR_k

Intuitively, when $E_k(m) < 0$, CR_k 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 CR_k is to reduce the error $E_k(m)$ to 0 and achieve the correct delay ratio between class *k* and *k*-1. Similar to the Absolute Delay Controller, the Relative Delay Controller also uses PI (Proportional-Integral) control (Equation (1)) to compute the control input (note that the parameters and variables are interpreted differently in the Absolute Delay Controller and the Relative Delay Controller).

For a system with *N* service classes, the Absolute Delay Guarantee is enforced by *N*-1 relative Delay Controllers CR_k ($1 \le k < N$). At every sampling instant *m*, the system calculates the process budget $B_k(m)$ of each class *k* as follows.

```
control_relative_delay ($\{W_k \mid 0 \le k < N\}$, $\{C_k(m) \mid 0 \le k < N\}$)
{
    Set class (N-1)'s process proportion $P_{N-1}(m) = 1$;
    $S = P_{N-1}(m)$;
    for ( $k = N-2$; $k \ge 0$; k--) {
        Calls $CR_{k+1}$ to get the process ratio $U_{k+1}(m)$ between class k and k+1;
        The process proportion of class k $P_k(m) = P_{k+1}(m)U_k(m)$
        $S = S + P_k(m)$;
    }
    for ( $k = N-1$; $k \ge 0$; k--)
        $B_k(m) = M (P_k(m) / S)$
}
```

4.4.3. The Hybrid Delay Controllers

The hybrid delay guarantee described in Section 3 can be implemented via dynamic switching between the Absolute Delay Controllers and the Relative Delay Controllers. The server switches from Relative Delay Controllers to Absolute Delay Controllers if the absolute delay guarantee of the highest priority class is violated, i.e., $C_0(m) > W_0 + H$; On the other hand, the server switches from Absolute Delay Controllers to Relative Delay Controllers if $C_0(m) < W_0 - H$. The use of a threshold window $\pm H$ in the mode switching condition is to avoid thrashing between the two sets of Controllers. Since the hybrid delay guarantee is a straightforward extension of absolute and relative delay guarantees, we focus on the design and evaluation of absolute and relative delay guarantees in the rest of this paper.

In summary, we have presented a feedback control architecture to achieve absolute, relative and hybrid delay guarantees on web servers. A key component in this architecture is the Controllers, which are responsible of dynamically computing correct process budgets in face of unpredictable workload and system variations. In the rest of the paper, we use the *closed-loop server* to refer to the adaptive web server with the Controllers (Figure 1), while the *open-loop server* refers to a non-adaptive web server without the Controllers. We present the design and tuning of the Controllers in the next section.

5. Design of the Controller

In this Section, we apply a control theory framework [23][33] to design the Relative Delay Controller CR_k and the Absolute Delay Controller CA_k. In Section 5.1, we specify the performance requirement of the Controllers. We then use system identification techniques to establish dynamic models for the web server in Section 5.2. Based on the dynamic model, we use the Root Locus method to design the Controllers that meet the performance specifications (Section 5.3).

5.1. Performance Specifications

In [34], we presented a set of performance metrics to characterize the performance of adaptive real-time systems based on control theory [27]. Compared with traditional metrics that only describe steady state performance, the specifications and metrics presented in [34] 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 specifications of the closed-loop server include in following.

- **Stability**: a (BIBO) stable system should have bounded output in response to bounded input. To the Relative Delay Controller (with a finite desired delay ratio), stability requires that the delay ratio should always be bounded at run-time. To the Absolute Delay Controller, stability requires that the service delay should always be bounded at run-time. Stability is a necessary condition for achieving desired relative or absolute delays.
- **Settling time T_s** is the time it takes the output to converge to the vicinity of the reference and enter 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 delay. As an example, we assume that our web server requires the settling time $T_s < 5$ min.
- **Steady state error E_s** is the difference between the reference input and average of output in steady state. The steady state error represents the accuracy of the Relative Delay Controller or Absolute Delay Controller in achieving the desired relative or absolute delays. As an example, we assume that our web server requires a steady state error $|E_s| < 0.1V_S$. Note that satisfying this steady state error requirement means that our web server can achieve the desired relative or absolute delays in steady state.

5.2. System Identification: Establishing Dynamic Models

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 the Controller. From the perspective of a Relative Delay Controller CR_k, the (control) input of the controlled system is the process ratio $U_k(m) = B_{k-1}(m)/B_k(m)$. The output of the controlled system is the delay ratio $V_k(m) = C_k(m)/C_{k-1}(m)$ (see Table 2). From the perspective of an Absolute Delay Controller CA_k, the (control) input of the controlled system is the process budget $U_k(m) = B_k(m)$. The output of the controlled system is the delay $V_k(m) = C_k(m)$ (see Table 1). We intentionally use the same symbols for input and output for Relative and Absolute Delay Controllers because the design methodology described below applies to both cases. Assuming the controlled system models for different classes are similar, we skip the class number k of $U_k(m)$ and $V_k(m)$ in the rest of this Section.

Unlike traditional control theory applications such as mechanical and electronic systems, it is often 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 controlled system (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 input [37] and use a least squares estimator to estimate the model parameters. Our experimental results (Section 7.2) established that, For both relative and absolute delay control, the controlled system 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.

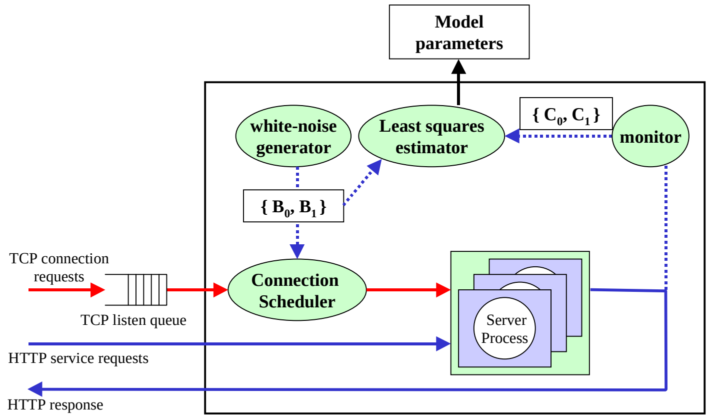


Figure 2 Architecture for system identification

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 budget 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 pseudo-random digital white noise generator to randomly switch two classes' process budgets between two configurations. White noise input has been commonly used for system identification [8]. The white noise algorithm is not presented due to space limitations. A standard algorithm for white noise can be found in [37].

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 for 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). 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 can "predict" a value $V^p(m)$ of the output by substituting the current estimates $\theta(m)$ into Equation (2). The difference $V(m)-V^p(m)$ between the measured output 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 system identification results (Section 7.2) established that, the controlled system can be modeled as a second order difference equation,

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

In the case of relative delay control, *V(m)* denotes the delay ratio between the two controlled classes, and *U(m)* denotes the process ratio between the two controlled classes, and the estimated model parameters are (Section 7.2):

$$(a_1, a_2, b_1, b_2) = (0.74, -0.37, 0.95, -0.12) \quad (6b)$$

In the case of absolute delay control, *V(m)* denotes the delay of one controlled class, and *U(m)* denotes the process budget of the controlled class. The estimated model parameters based on system identification experiments (Section 7.2) are

$$(a_1, a_2, b_1, b_2) = (-0.08, -0.2, -0.2, -0.05) \quad (6c)$$

5.3. Root-Locus Design

Given a model described by Equation (6a), we can apply control theory methods such as the Root Locus [27] to design the Relative Delay Controller and the Absolute Delay Controller. The controlled system model in Equation (6a) 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 controlled system model and the Controller model, the transfer function of the closed loop system 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 use the Root Locus tool of MATLAB [28] 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 [27].

To design the Relative Delay Controller, we use the Root Locus tool to plot the traces of the closed loop poles (based on the model parameters in Equation (6b)) as the controller gain increases are illustrated on the z-plane in Figure 3. The closed-loop poles are placed at

$$p_0 = 0.70 \qquad p_{1,2} = 0.38 \pm 0.62i \tag{10a}$$

(see Figure 3) by setting the Relative Delay Controller's parameters to

$$g = 0.3 \qquad r = 0.05 \tag{10b}$$

Similarly, to design the Relative Delay Controller (based on the model parameters in Equation (6c)), the closed-loop poles are placed at

$$p_0 = 0.607 \qquad p_{1,2} = -0.30 \pm 0.59i \tag{11a}$$

by setting the Absolute Delay Controller's controller parameters to

$$g = -4.6 \qquad r = 0.3 \tag{11b}$$

The above pole placement is chosen to achieve the following properties in the closed loop system [27]:

- **Stability**: The closed-loop system with the Relative Delay Controller (with parameters in Equation (10b)) or the Absolute Delay Controller (with parameters in Equation (11b)) guarantees stability because all the closed-loop poles are in the unit circle, i.e., $|p_j| < 1$ ($0 \leq j \leq 2$) (Equations (10a) and (11a)).
- **Settling time**: According to control theory, decreasing the radius (i.e., the distance to the origin in the z-plane) of the closed-loop poles usually results in shorter settling time. The Relative Delay Controller (with Equation (10b)) achieves a settling time of 270 sec, and the Absolute Delay Controller (with Equation (11b)) achieves a settling time of 210 sec, both lower than the required settling time (300 sec) defined in Section 5.1.
- **Steady state error**: Both the Relative Delay Controller and the Absolute Delay Controller achieve zero steady state error, i.e., $E_s = 0$. This result can be easily proved using the Final Value Theorem in digital control theory [23]. This result means that, in steady state, the closed-loop system with the Relative Delay Controller or the Absolute Delay Controller guarantees the desired relative delays or the desired absolute delays, respectively.

In summary, using feedback control theory techniques including system identification and the Root Locus design, we systematically design the Relative Delay Controller and the Absolute Delay Controller that *analytically* provide the desired relative or absolute delay guarantee and meet the transient and steady state performance specifications described in Section 5.1. This result shows the strength of the control-theory-based design framework for adaptive computing systems.

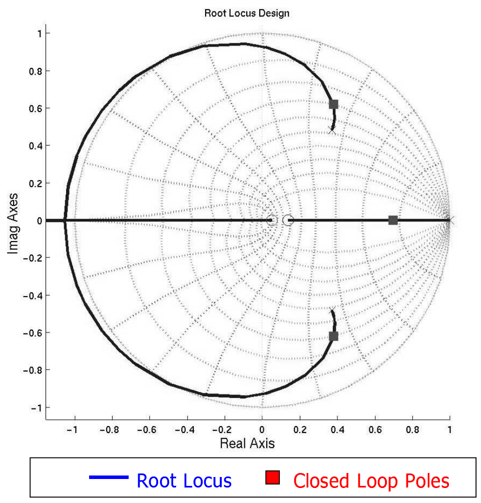


Figure 3 The Root Locus of the web server model

6. Implementation

We now describe the implementation of the web server. 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 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 (modified from Apache). 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 [6] 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 (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 then passed to the Controllers, which implements the control algorithm to compute new process budgets.

- 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 Controllers. An open-loop server can be configured for either system identification or performance evaluation.

[6] Other criteria for connection classification include HTTP cookies, Browser plug-ins, URL request type or filename path, and destination IP address of virtual servers [12].

7. Experimentation

All experiments were conducted on a testbed of ten 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 was as follows.

- **Client:** We used SURGE [13] to generate realistic web workloads in our experiments. 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 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. 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 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 time of a server with HTTP 1.1. The experiments on system identification are presented in Section 7.2. We present the evaluation of the closed-loop server in Section 7.3.

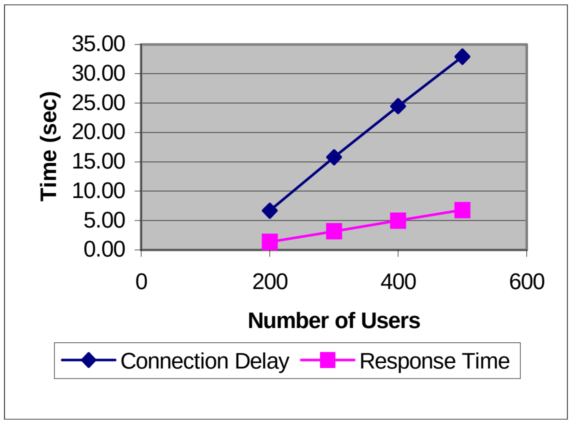
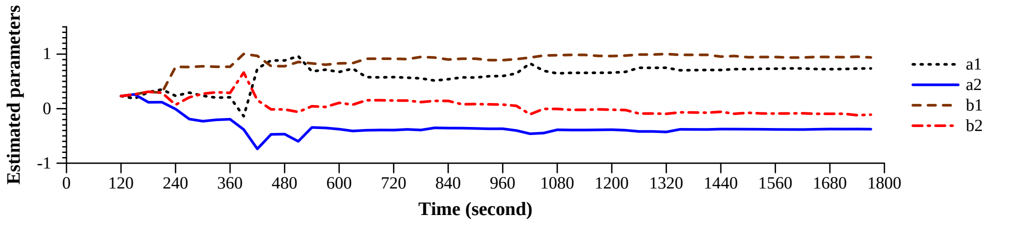


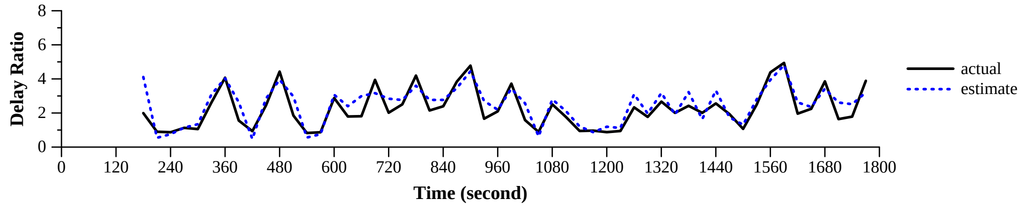
Figure 4 Connection delay and response time

7.1. Comparing Connection Delays and Response Times

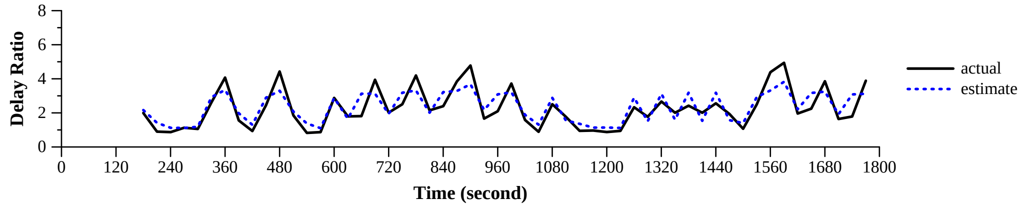
In the first set of experiments, we compare the average connection delay and the average response time (per HTTP request) of an open-loop server (see Figure 4) to justify the use of connection delay as a metric for service differentiation in web servers with HTTP 1.1. All connections are treated as being in a same class and all server processes are allocated to the class. 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 at a much faster rate than the response time as the number of users increases. For example, when the number of users is 400, the connection delay is 4.9 times the response time. Note that the average response time is computed based on two types of requests, i.e., the response time (including the connection delay and the processing delay) of the first request of each connection and the response time (including only the processing time) of each subsequent request. The difference between connection delay and response time is due to the fact that processing delay is on average significantly shorter than connection delay. We also run similar experiments with 256 server processes (the maximum number allowed by the original Apache on Linux). With 256 server processes, the ratio between the connection delay and the response time is similar to that presented in Figure 4. For example, the connection delay was 5.3 times the response time when 400 users are used. The complete result for this case is not presented due to space limitations. This result justifies our decision to use connection delay as a metric for service differentiation in web servers with HTTP 1.1.



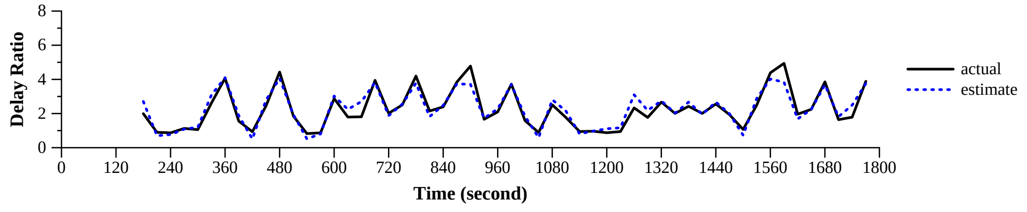
(a) Estimated model parameters (second order model)



(b) Modeling error (second-order model)



(c) Modeling error (first-order model)



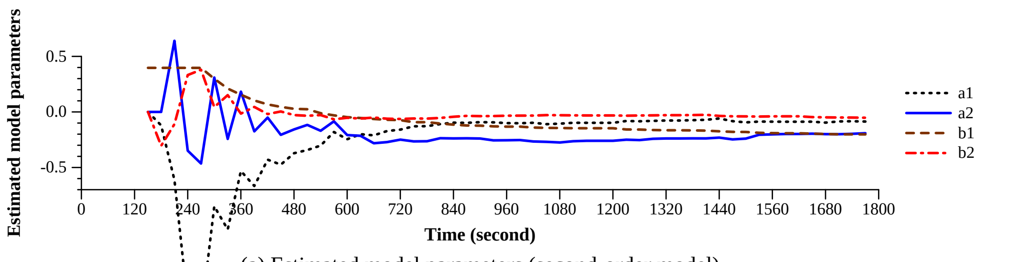
(d) Modeling error (third-order model)

Figure 5 System identification results for Relative Delay

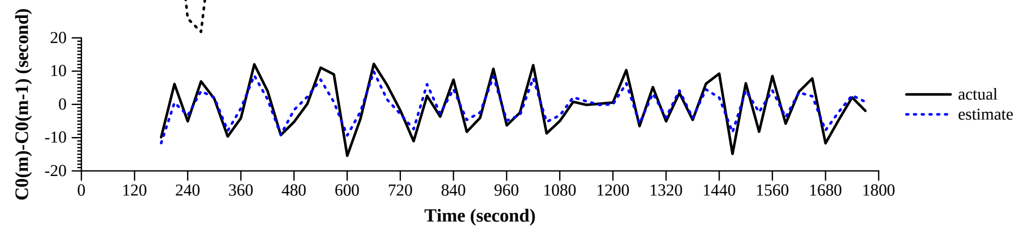
7.2. System Identification

We now present the results of system identification experiments for both relative delay and absolute delay to establish a dynamic model for the open-loop system. Four client machines are divided into two classes 0 and 1, and each class has 200 users. We begin with the relative delay experiments. The input, process ratio $U(m) = B_0(m)/B_1(m)$, is initialized to 1. At each sampling instant, the white noise randomly sets the process ratio to 3 or 1. The sampled output, the relative delay $V(m) = C_1(m)/C_0(m)$ is fed to the least squares estimator to estimate model parameters (Equation (2)). Figure 5(a) shows that the estimated parameters of a second order model (Equation (6)) at successive sampling instants in a 30 min run. The estimator and the white noise generator are turned on 2 min 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) converge 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-run the experiment with a different white noise input (i.e., with a different random seed) to the open-loop server and compare 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 throughout the 30 min run. This result shows that the estimated second order model is adequate for designing the Relative Delay Controller. We also re-ran the system identification experiments to estimate a first order model and a third order model. The results demonstrate that the estimated first order model had larger prediction error than the second order model (see Figure 5(c)), while an estimated third order model does not tangibly improve the modeling accuracy (see Figure 5(d)). Hence the second order model is chosen as the best compromise between accuracy and complexity.

The system identification experiments are repeated with the same workload and configurations for the absolute delay. The input of the open loop system is the process budget $U(m) = B_0(m)$ of class 0, which is initialized to 64. At each sampling instant, the white noise randomly sets the process budget to 96 or 64. The output is the sampled delay $V(m) = C_0(m)$ of class 0. To linearize the model, we feed 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 in Equation (2). Figure 6(a) shows that the estimated parameters of the second order model (Equation (6)) at successive sampling instants in a 30 min run. The estimations of the parameters (a_1, a_1, b_1, b_2) converge to (-0.08, -0.2, -0.2, -0.05). To verify the accuracy of the model, we re-run the experiment with a different white noise input to the open-loop server and compare the actual difference between two consecutive delay samples with that predicted by the estimated model (Figure 6(b)). Similar to the relative delay case, the prediction of the estimated model is consistent with the actual delay throughout the 30 min run. This result shows that the estimated second order model is adequate for designing the Absolute Delay Controller.



(a) Estimated model parameters (second-order model)



Modeling error (second-order model)

Figure 6: System Identification Results for Absolute Delay

7.3. Evaluation of the Adaptive Web Server

In this section, we present evaluation results for our adaptive web server. In Section 7.3.1, we first present the evaluation results of the Relative Delay Controller. The results for guaranteeing the relative delays of a server with three classes are presented in Section 7.3.2. The evaluation results of absolute delay guarantee are presented in Section 7.3.3.

7.3.1. Evaluation of Relative Delay Guarantees between Two Classes

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

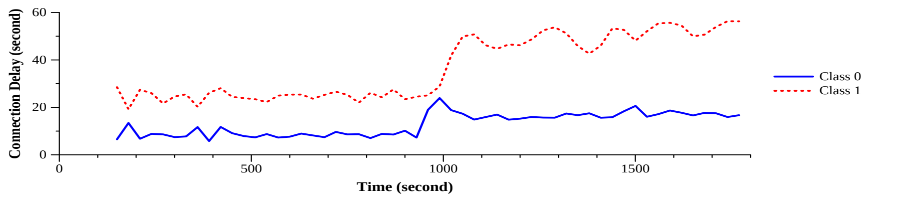
- **Workload**: Four client machines are evenly divided into two classes. Each client machine has 100 users. In the first half of each run, only one client machine from class 0 and two client machines from class 1 (100 users from class 0 and 200 users from class 1) generate HTTP requests to the server. The second machine from class 0 starts generating HTTP requests 870 sec later than the other three machines. Therefore, the user population changes 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 is W_1/W_0 = 3. The process ratio $B_0(m)/B_1(m)$ is initialized to 1 in the beginning of the experiments. To avoid the starting phase of SURGE, the Controller is turned on 150 sec after SURGE started. The sampled absolute connection delays and the delay ratio between the two classes are illustrated in Figure 7(a) and (b), respectively.
- **Open-loop server**: An open-loop server is also tested as a baseline. The open-loop server is 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 Figure 7(c)(d).

We first look at the first half of the experiment on the closed-loop server (Figure 7(a)(b)). When the Controller is turned on at 150 sec, the delay ratio $C_1(m)/C_0(m)$ = (28.5 sec / 6.5 sec) = 4.4 due to incorrect process allocation. The Controller dynamically reallocates processes and changes the relative delay to the vicinity of the reference W_1/W_0 = 3. The relative delay stays 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. 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 often face with unpredictable and bursty workload [20]. The robustness of our closed-loop server is demonstrated by its response to the load variation starting at 870 sec (Figure 7(a)(b)). Because the number of users of class 0 suddenly increases from 100 to 200, the delay ratio drops from 3.2 (at 870 sec) to 1.2 (at 900 sec) - far below the reference W_1/W_0 = 3. The Controller reacts 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-converges to 2.9.

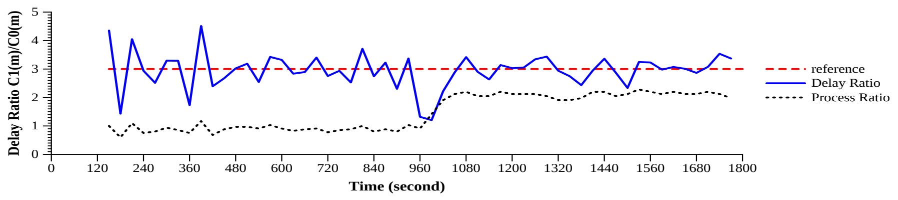
In contrast, while the open-loop server achieves satisfactory relative delays when the workload conforms to its expectation (from 150 sec to 900 sec), it violates the relative delay guarantee after the workload changes (see Figure 7(c)(d)). After the workload changes (from 960 sec to the end of the run), connections from class 0 consistently have longer delays than connections from class 1.

In terms of the control metrics, the closed-loop server maintains stability because its relative delay is clearly bounded throughout the run. We observe from (Figure 7(b)) that the server renders satisfactory efficiency and accuracy in achieving the desired relative delays. In particular, in response to the workload variation at time 870 sec, the duration of the distinguishable performance deviation from the reference lasts for 180 sec (from 900 sec to 1080 sec), well within the theoretical settling time of 270 sec based on our design (Section 5.3). The delay ratio stays close to the reference in steady state, which demonstrates a small steady state error[7].

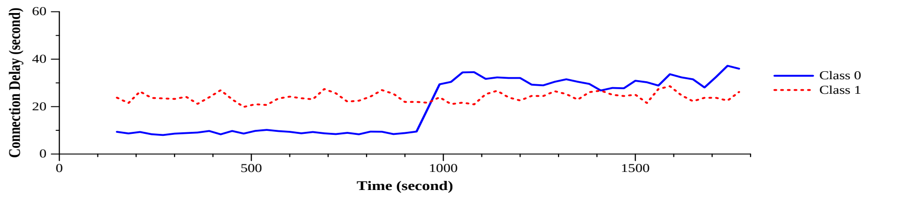
[7] Due to the noise of the server caused by the random workload, it is impossible to precisely quantify the settling time and steady state error based on the ideal definitions (Section 5.1).



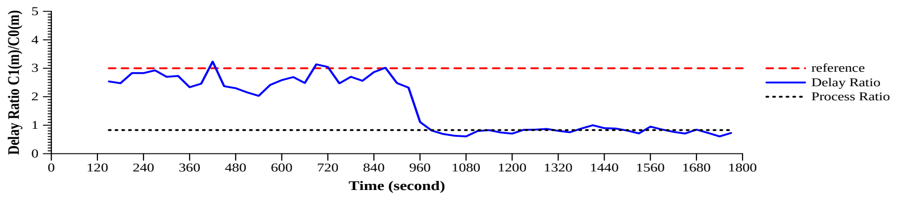
(a) Close-loop: Connection Delays C(0) and C(1)



(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 7: Evaluation Results of Relative Delay Guarantees between Two 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 has a client machine with 100 users. The Controller is turned on at 150 sec. The desired relative delays are (W_0, W_1, W_2) = (1, 2, 4). The process proportions are initialized to (P_0, P_1, P_2) = (1, 1, 1). From Figure 8, we can see that the connection delay begin at (C_0, C_1, C_2) = (14.6, 17.3, 17.5) which has the ratio (1, 1.2, 1.2), and then changes to (C_0, C_1, C_2) = (9.3, 16.2, 33.9) which has the ratio (1, 1.7, 3.6), i.e., close to the desired relative delay, 240 sec after the Controller is turned on. The relative connection delay remains bounded and close to the desired relative delay in steady state. This experiment demonstrates the Relative Controllers can guarantee desired relative delays for more than two classes.

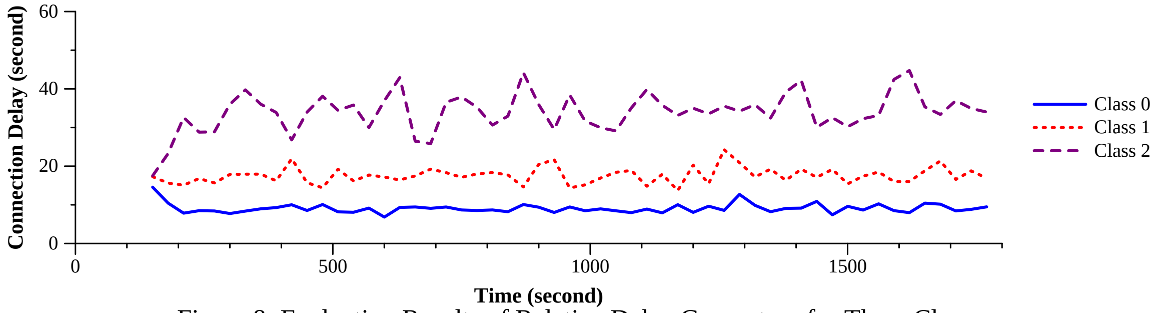


Figure 8: Evaluation Results of Relative Delay Guarantees for Three Classes

7.3.3. Evaluation of Absolute Delay Guarantees

In this section, we evaluate the absolute delay guarantee for two classes. The experiment is set up as follows.

- **Workload**: The same workload described in Section 7.3.1 is used to evaluate the absolute guarantees. In the first half of each run, 100 users from class 0 and 200 users from class 1 generate HTTP requests to the server. Another 100 users from class 0 start generating HTTP requests 870 sec later than the original users. Thus the user population changes 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 delays for class 1 and 0) to the Controller is (W_0, W_1) = (10, 30) (sec). The process budgets ($B_0(m)$, $B_1(m)$) are initialized to 64 for each class in the beginning of the experiments. To avoid the start up phase of SURGE, the Controller is turned on 150 sec after SURGE started. The sampled absolute connection delays of the two classes are illustrated in Figure 9(a).
- **Open-loop server**: An open-loop server is tested as a baseline. The open-loop server is fine-tuned to have a "correct" process allocation to achieve the desired absolute delays 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 Figure 9(b).

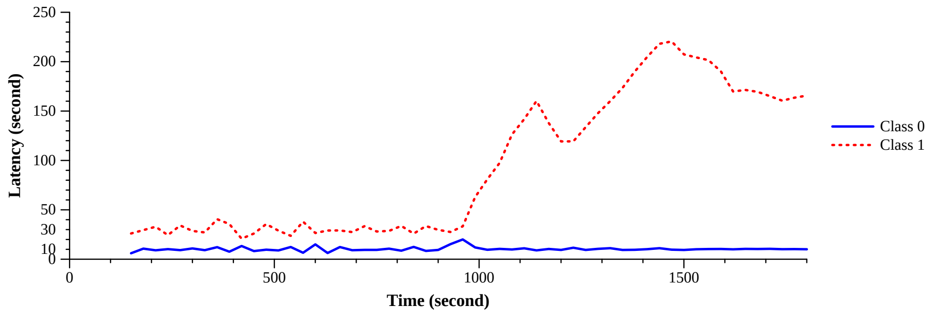
In the first half of the experiment on the closed-loop server (Figure 9(a)), the Controllers dynamically allocate processes and the delays of both classes remain close to their desired delay (10 sec and 30 sec, respectively). At time 870 sec, the number of users of class 0 suddenly increases from 100 to 200, and the delay of class 0 increases from 8.4 sec (at time 870 sec) to 20.0 sec (at time 900 sec) – violating its absolute delay guarantee (10 sec). The Controllers react to the load variation by allocating more processes to class 0 and decreasing the number of processes allocated to class 1. By time 1020 sec, the delay of class 0 successfully re-converges to 9.6 sec at the cost of violating the delay guarantee of the low priority class (class 1)[8].

In comparison, while the open-loop server achieves satisfactory delays for both classes when the workload is similar to its expectation (from 150 sec to 900 sec), it fails to provide delay guarantee for class 0 with the highest priority, after the workload changes (see Figure 9(b)). Instead, connections from class 0 consistently have longer delays than connections from class 1 after the workload changes, i.e., the open-loop server fails to achieve the desired delay for the high priority class.

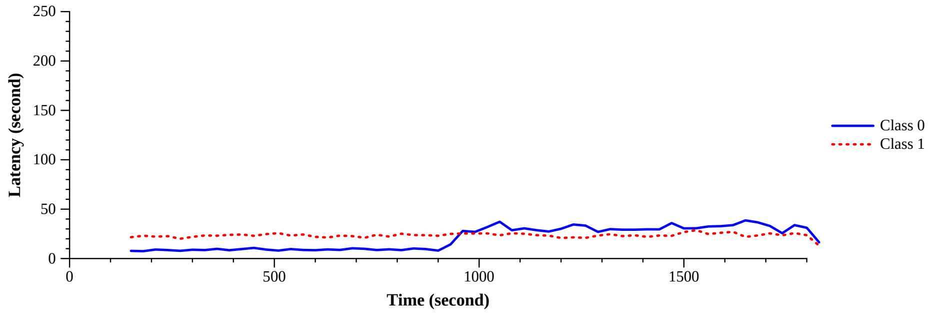
Note that while both the open loop server and the closed loop server violate the delay guarantee of one service class, the closed loop server provides the correct order of guarantee violation by discriminating against the low priority class, while the open loop server fails to achieve the correct order. In terms of control metrics, the unsaturated (high priority class) controller maintains stability because its delay is clearly bounded throughout the run. Note that because the system load can grow arbitrarily, Absolute Delay Controllers (especially those of low priority classes) can saturate and becomes unstable in overload conditions even if it is tuned correctly. We observe from (Figure 9(a)) that the server renders satisfactory efficiency and accuracy in achieving the desired delay for the high priority class (class 0). In particular, in response to the workload variation at time 870 sec, the duration of the distinguishable performance deviation from the reference lasts for 60 sec (from 930 sec to 990 sec), well within the theoretical settling time of 210 sec based on the control design (Section 5.3). The delay of class 0 stays close to the reference in steady state, which demonstrates a small steady state error for high priority class,

[8] Note that the low priority class suffers extremely long service delay in the second half of the closed loop experiment. In such overload conditions, the system devotes most processes to high priority classes to provide their absolute delay guarantees, and consequently starves low priority classes. This situation is unavoidable in any servers that provide absolute guarantees.

i.e., the desired delay of the high priority class is guaranteed in steady state even when the server is severely overloaded.



(a) Connection Delays of the Closed Loop Server



(b) Connection Delays of the Open Loop Server

Figure 9: Evaluation of Absolute Delay Guarantees

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

8. Related Work

Support for different classes of service on the web (with special emphasis on server delay differentiation) has been investigated in recent literature. For example, the authors of [23] proposed and evaluated an architecture in which restrictions are imposed on the amount of server resources (such as threads or processes) which are available to basic clients. In [5][6] admission control and scheduling algorithms are used to provide premium clients with better service. In [11] 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 and hence can be called the *best effort differentiation* model.

Several other works such as [10][22][30] developed kernel level mechanism to achieve overload protection and proportional resource allocations in server systems. Their work also did not provide relative or absolute delay guarantees in web servers. Supporting proportional differentiated services in network routers have been investigated in [21][31]. Their work did not address end systems such as web servers.

There have been several results that applied feedback control theory to the design of real-time computing systems. For example, several papers [11][16][17][23][36][43][44] focused on adaptive real-time (CPU) scheduling techniques to improve digital control system performance by exploiting the elastic timing constraints in such systems. These techniques are tailored to the specific characteristics of digital control systems instead of web servers. Transient and steady state performance of adaptive real-time systems has received special attention in recent years (e.g., [15][40][45]). For example, Brandt et. al. [15] evaluated a dynamic QoS manager by measuring the transient performance of applications in response to QoS adaptations. Rosu et. al. [40] proposed a set of performance metrics to capture the transient responsiveness of adaptations and its impact on applications. In [31], Li et. al. applied control theory based techniques to achieve the desired throughput over the network in a distributed visual tracking system. System delay and web servers were not addressed in their work. Adaptive QoS management architectures (e.g., [2][4][9][29][39][46]) have been developed to support applications such as communication, multimedia and embedded systems. However, these architectures were not designed based on a unified theoretical framework such as control theory.

A least squares estimator was used in [1] for automatic profiling of resource usage parameters of a web server. However, the work was not concerned with establishing a dynamic model for the server. In [3][5], a feedback control loop was used to control the desired CPU utilization of a web server with adaptive admission control. By controlling the CPU utilization, the CPU utilization control can guarantee the desired absolute delay in web servers under HTTP 1.0 protocol and when CPU is the bottleneck resource. This technique is not applicable to servers under HTTP 1.1 protocol. In [34], we proposed a control-theory-based design framework for adaptive real-time systems to guarantee low deadline miss-ratio in unpredictable environments. This paper extends the framework in [34] to web servers for guaranteeing desired relative service delays among service classes.

9. Conclusion and Future Work

In this paper, we present the design and implementation of an adaptive architecture to provide relative, absolute and hybrid service delay guarantees for different service classes on web servers under HTTP 1.1. The first contribution of this paper is the architecture based on feedback control loops that enforce delay guarantees for different 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 dynamic models for a web server, and using the Root Locus method to design feedback controllers to satisfy performance specifications. The adaptive architecture has been implemented by modifying an Apache web server. Experimental results demonstrate that our adaptive server provides robust delay guarantees even when workload varies significantly. Properties of our adaptive web server also include guaranteed stability, and satisfactory efficiency and accuracy in achieving desired delay or delay differentiation. In the future, we will extend our architecture to provide QoS guarantees in networked embedded systems and web server farms.

Acknowledgements

The authors would like to thank Gang Tao, John Regehr and Jorg Liebeherr 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, Arizona, December 1999, pp. 44-53.

[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] C. Aurrecoechea, A. Cambell, and L. Hauw, "A Survey of QoS Architectures," *4th IFIP International Conference on Quality of Service*, Paris, France, March 1996.

[10] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," *Operating Systems Design and Implementation (OSDI'96)*, 1999.

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

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

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

[14] 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.

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

[16] G. Buttazzo, G. Lipari, and L. Abeni, "Elastic Task Model for Adaptive Rate Control", *IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 286-295, December 1998.

[17] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity Sharing for Overrun Control," *IEEE Real-Time Systems Symposium*, Orlando, FL, December 2000.

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

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

[20] 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.

[21] C. Dovrolis, D. Stiliadis, and P. Ramanathan, "Proportional Differentiated Services: Delay Differentiation and Packet Scheduling," *SIGCOMM'99*, Cambridge, Massachusetts, August 1999.

[22] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," *Operating Systems Design and Implementation (OSDI'96)*, Seattle, WA, October 1996.

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

[24] 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.

[25] E-Soft Inc., "Web Server Survey," http://www.securityspace.com.

[26] 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.

[27] G. F. Franklin, J. D. Powell and A. Emami-Naeini, *Feedback Control of Dynamic Systems (3rd Ed.)*, Addison-Wesley, 1994.

[28] Mathworks Inc., http://www.mathworks.com/products/matlab.

[29] D. Hull, A. Shankar, K. Nahrstedt, and J. W. S. Liu, "An End-to-End QoS Model and Management Architecture," *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, Dec 1997.

[30] 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*, Madrid, Spain, December 1998.

[31] 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.

[32] J. Liebeherr and N. Christin, "Buffer Management and Scheduling for Enhanced Differentiated Services," *University of Virginia Tech. Report CS-2000-24*, August 22, 2000.

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

[34] 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.

[35] 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.

[36] L. Palopoli, L. Abeni, F. Conticelli, M. D. Natale, and G. Buttazzo, "Real-Time control system analysis: An integrated approach," *IEEE Real-Time Systems Symposium*, Orlando, FL, Dec 2000.

[37] 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, pp. 1192-1201.

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

[39] 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.

[40] 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.

[41] M. Ryu and S. Hong, "Toward Automatic Synthesis of Schedulable Real-Time Controllers", *Integrated Computer-Aided Engineering*, 5(3) 261-277, 1998.

[42] 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.

[43] 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.

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

[45] L. R. Welch and B. A. Shirazi, "A Dynamic Real-time Benchmark for Assessment of QoS and Resource Management Technology," *IEEE Real-time Technology and Applications Symposium*, June 1999.

[46] 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.