

# Autonomous Delay Regulation for Multi-Threaded Internet Servers

Jin Heo, Xue Liu, Lui Sha, Tarek F. Abdelzaher  
{jinheo, xueliu, lrs, zaher}@cs.uiuc.edu

Department of Computer Science, Univ. of Illinois at Urbana-Champaign

## Abstract

*How to properly assign system resources to satisfy the Service Level Agreement (SLA) is a challenging research problem. Previous approaches include queuing model based feedback control strategies with queuing predictor for feed-forward delay prediction. However, ignorance of the multi-threaded nature can induce large model errors. To compensate this, previous approaches typically perform offline identification, thus making the system dependent on a particular workload and a specific Internet application.*

*In this paper, we propose a novel framework for autonomous delay regulation for multi-threaded Internet servers. We formulate a processor-sharing queuing model for the multi-threaded server architecture to precisely predict service rate for worker threads. In addition, the proposed scheme uses the sleep actuator to properly assign resources based on the calculated service rate. We evaluate our techniques experimentally using an Apache web server test-bed. We demonstrate that the proposed strategy performs better than the previous approaches under a realistic workload.*

## 1. Introduction

As Internet services have become an integral part of our information services infrastructure, delay regulation of individual request to an Internet service is a challenging problem with important practical implications. Too short response time will cause lowered resource usage, hence reduced revenues. Additionally, too large delay also reduces revenue. The objective of the delay regulation of an Internet application is to meet the response time specified in SLA (Service Level Agreement) [1] while maximizing revenue.

Recently, feedback control theory has been applied to provide performance guarantees in computing systems such as Internet servers [3] and communication systems. However, since a computing system's response to allocated resources is highly non-linear, differential equation models used by control theory does not abstract computing systems well. Starting from this perspective, in [21], Sha et al. presents Queuing Model-Based Feedback Control to integrate queuing theory with control theory. At the core of the design is a model-based feed-forward predictor, which keeps the system state near an equilibrium operation point in presence of dynamic workloads. This essentially linearizes the system. Combined with a feedback PI controller to

suppress the “residual errors”, the Queuing Model Based Feedback Control architecture is shown to provide good mean delay regulation in both simulations and instrumented Apache Web server systems [5]. Single queuing models such as M/M/1 or M/G/1 are popular for modeling Internet servers and used in [16][21]. For resource allocation, thread-based (or process based) allocation mechanisms [9][8][14][21][23] are typically used, where the number of worker threads or processes is adjusted based on the calculated control output (e.g. service rate).

Two problems are often encountered implementing these approaches. First, single queue queuing models likes of M/M/1 inherently lack of capturing the effect of the multi-threaded nature of Internet servers. Queuing predictors using those models produce service rate for the entire system not for worker threads. Thus, how to properly allocate resources in a multi-threaded Internet server using single queue models is not a simple matter. Secondly, the performance of thread-based allocation mechanism relies on the fact that the increase in the number of worker threads enlarges concurrency gain, thus leading to improvement of performance of a system. However, the extent to which thread-based mechanisms are effective depends heavily on the degree of interaction between the running threads, which further depends on the nature of the workload of incoming requests [23]. To remedy these problems, off-line system identification is often used to find the precise relationship between thread allocation and the control output [16][9][8][14][21][23]. Nonetheless, the performance still remains tied to a specific workload and an implementation.

In this paper, we present an autonomous delay regulation technique for multi-threaded Internet servers. We first derive a queuing formula for precise prediction of service rate in multi-threaded architecture and present the sleep actuator which regulates the service rate by enforcing worker threads to sleep for certain amount of time. To demonstrate the performance of the proposed scheme, we set up a test-bed and implement a delay regulation framework for Apache Web server, one of the most widely used multi-threaded Internet servers.

Our approach is distinguished through several features:

- *Capturing multi-threaded nature:* By deriving a solution to calculate service rate for a worker thread, queuing model predictor can perform more accurately, thus reduce residual error and response time fluctuation.
- *Self-Regulating:* The proposed scheme does not need any prior identification phase. Hence, the performance is independ-

ent to workload changes and specific characteristics of Internet servers.

The rest of the paper is organized as follows. Section 2 provides the background of the multi-threaded Internet server architecture and Queuing Model-Based Feedback Control. In Section 3, the proposed Enhanced Queuing Model-Based Feedback Control is presented. In Section 4, we present the validation procedure by implementing the Enhanced Queuing Model-Based Feedback Control on Apache Web server. Related work is discussed in Section 5 and finally, we conclude the paper with directions of future work in Section 6.

## 2. Background

In this section we first present HTTP processing steps and the architecture of a Web server since HTTP protocols and Web servers are one of the most popular protocols and applications. We believe other Internet server applications using TCP connections to receive client requests such as database servers and FTP servers have a similar architecture to the one explained here. After that we briefly introduce queuing Model-Based Feedback Control. More interested users can refer to [21].

### 2.1. HTTP Background

Generally, HTTP servers have one listen thread waiting for client requests on a well know TCP port, which is typically 80 and several worker threads. A new TCP connection is initiated by a client, which sends a TCP SYN packet. This packet is received on the listening socket of the HTTP server. On receiving TCP SYN, the server responds with a SYN-ACK packet to acknowledge that it received the TCP SYN packet and create a temporary structure which represents this new TCP connection request and placed it in the SYN-RCVD queue. After the client responds with an ACK packet to the SYN-ACK packet, the server extracts the TCP request from the SYN-RCVD queue, put it into the accept queue of the listen socket and wait for the listen thread to call `accept()` [4][3]. After the listen thread calls `accept()` system call, then, finally a TCP connection is made. At this time, the listen thread dispatches the request to one of available worker threads [5]. This assignment of new worker thread can be done by pulling an idle thread out of a worker thread pool or forking a new thread. For example, Apache uses worker thread pool to decrease thread creation overhead. After a worker thread is assigned a request by the listen thread, it handles the request and sends a response back to the client. The following represents the basic sequential steps of HTTP request processing of a worker.

- *Read Requests and Parsing*: Once a request comes in a HTTP worker, it needs to be prepared to retrieve a proper Web object.
- *Read Files or Generate dynamically*: After identifying which object to retrieve, a worker reads it either from storages or memory cache if is a static page. If not, it generates dynamically.

- *Send Response*: After reading the object, a worker sends it through a network interface

Hence, the total time to process a request in a web server (delay) consist of

- *TCP Negotiation Time*: TCP uses three-way handshake protocol to set up a connection
- *Queuing Delay in Accept Queue*: After connection is made, new socket structure and descriptor is created and put into the accept queue(listen queue) before `accept()` is called to extract the socket descriptor.
- *HTTP Processing Time (Service Time)*: Each HTTP worker handles one request each time.

In this paper, we focus on adjusting HTTP processing time to achieve delay regulation specified in SLA. However, we would like to mention that the framework suggested in this paper can be easily combined with admission control [33] to protect a system from overload.

## 3. Autonomous Delay Regulation for multi-threaded Internet Servers

The objective of delay regulation is to keep the delay as close as possible to the reference value specified by Service Level Agreement (SLA). If delays exceed the reference, it is unacceptable to the user. In the opposite, too short delays than the reference leads to inefficient use of resources and less revenue.

In this section, we present a novel autonomous delay regulation framework for multi-threaded Internet servers. The proposed autonomous delay regulation framework is differentiated from several aspects. First, we present an accurate server model for multi-threaded Internet servers. Since most Internet servers including Web servers exploit the multi-thread architecture explained in section 2.1, it will improve the performance of delay regulation if we have a precise model for multi-threaded Internet servers. To this end, we present a new queuing model. This new queuing model also greatly simplifies performance control, since the new model has the similar properties to M/M/1 model which is easy to implement and needs small overhead. Next, we propose a sleep actuator as to adjust the service rate (allocate resources) properly in a multi-threaded architecture instead of using the most popular type of actuators that change the number of worker threads to allocate resources. One of the drawbacks of this type of actuators is that the performance of the actuators is dependent on the characteristics of a specific workload and the application structure, hence making offline system profiling necessary. The sleep actuator proposed here is seamlessly integrated to our new queuing model, since it basically works at the thread level.

In what follows, we first derive a queuing formula to calculate the service rate of a worker thread in the multi-threaded Internet server and present the operation steps for the proposed approach. Afterwards, we explain the sleep actuator and its de-

sign considerations. Finally the design procedure for the feedback controller is described.

### 3.1. A Queuing Model For A Multi-Threaded Internet Server

Typically, when more worker threads are assigned to process the incoming requests, the degree of concurrency increases, yielding in increase in throughput. This is because as explained in section 2, both CPU and I/O handling (network I/O and file I/O) are drawn in processing HTTP requests implicating overlapping of thread executions. For example, if thread A is reading a HTTP request from socket, thread B is extracting the content of a file and thread C is executing a CGI script, those three threads can run concurrently. However, concurrency gain is in general not proportional to the increase in the number of worker threads due to the contention between threads and scheduling overheads. In addition, different types of workloads (e.g., I/O-bound and CPU-bound) can lead to different concurrency gain [23].

Rather than developing a complicated model for a multi-threaded Internet server to precisely model concurrency gain, we resort to Processor Sharing (PS) queuing model assuming no concurrency gain from the multi-threaded architecture. We do not consider concurrency gain or overhead. One possible way of modeling a Web server is M/G/1/PS or M/M/1 PS (Processor Sharing) queue [28]. In a single server PS queue, the capacity  $C$  of the server is equally shared between the customers in system; if there are  $n$  customers in system each receives service at the rate  $C/n$ . Customers do not have to wait at all and the service starts immediately upon arrival. However in a multi-threaded Internet server architecture, there is generally a thread limit for number of worker threads to assign for requests and if the number of requests exceeds it, a request should wait for the next available worker thread while sitting in the TCP accept queue. Hence, M/G/1 PS queuing system would not be adequate in that case.

Thus, we assume that only worker threads currently processing HTTP requests time-share CPU and again do not assume any concurrency gain from a multi-threaded architecture or overhead caused by it. This assumption holds when a workload is CPU-intensive or modestly I/O-intensive such that most of the requests can be retrieved from cache, which is reasonable since dynamic pages occupy a good portion of HTTP requests nowadays. If concurrency gain is high or overhead is not negligible, this assumption will yield model errors but we will demonstrate in later this paper that it still helps to linearize the system near the set point in the presence of realistic workload.

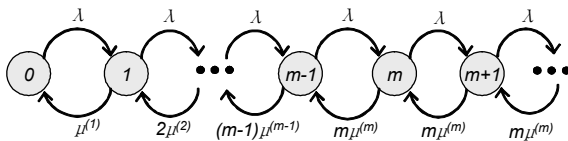


Figure 1. State-transition-rate diagram

With this assumption, we model a multi-threaded Internet server using Birth-Death Process, a variation of Continuous-Time Markov Chain [32]. The arrival process follows Poisson distribution and the service time is exponentially distributed. This assumption trades off accuracy of model, which can be corrected by a feedback controller. We will show later the proposed queuing model combined with a simple P controller exhibit good performance indeed. In this queuing model, at most  $m$  jobs arrived first in the queue receive an infinitesimal quantum of service in a Round-Robin (RR) fashion. When the quantum expires and if they need more service, they are suspended and wait for other  $m-1$  jobs to consume their quantum. When a job has received the amount of service required, it leaves the queue. Figure 1 shows the state-transition-rate diagram for the queuing system. State  $k$  represents where there are  $k$  jobs in the system.  $\mu_{system}^{(k)}$  represents state dependent system service rate and  $\mu^{(k)}$  is state dependent service rate of each thread, where there are  $k$  jobs being serviced.

$$\lambda_k = \lambda, \quad k = 0, 1, 2, \dots \quad (1)$$

$$\mu_{system}^{(k)} = \begin{cases} k\mu^{(k)} & (0 \leq k \leq m) \\ m\mu^{(m)} & (m \leq k) \end{cases} \quad (2)$$

From the Steady state solution of a birth death process [32], steady state probability  $P_k$  of being in state  $k$  is

$$P_k = \frac{\lambda_0 \lambda_1 \Lambda \lambda_{k-1}}{\mu_{system}^{(1)} \mu_{system}^{(2)} \Lambda \mu_{system}^{(k)}} P_0, \quad k = 1, 2, K, \infty \quad (3)$$

,where  $P_0$  is the probability of being in the initial state. Since threads share the CPU equally such that

$$\mu_{system}^{(1)} = \mu_{system}^{(2)} = \Lambda = \mu_{system}^{(k)} = \mu_{system}, \quad k = 1, 2, K, \infty,$$

it follows that

$$P_k = \left( \frac{\lambda}{\mu_{system}} \right)^k P_0 = \rho^k P_0, \quad k = 1, 2, K, \infty \quad (4)$$

where

$$\rho = \frac{\lambda}{\mu_{system}} = \frac{\lambda}{\min(m, k) \times \mu^{(k)}}$$

Steady state population of the system  $\bar{N}$  is

$$\bar{N} = \sum_{k=1}^{\infty} k P_k = \sum_{k=1}^{\infty} k (1 - \rho) \rho^k = \frac{\rho}{1 - \rho} \quad (5)$$

Using Little's formula [32], expected delay  $\bar{D}$  is

$$\bar{D} = \frac{\bar{N}}{\lambda} = \frac{1}{\mu_{system} - \lambda} = \frac{1}{\min(m, k) \times \mu^{(k)} - \lambda} \quad (6)$$

From (6), we can get the service rate for each thread when the number of active threads (threads which are currently serving a request) is  $k$ .

$$\mu^{(k)} = \frac{1}{\min(m, k)} \left( \frac{1}{D} + \lambda \right) \quad (7)$$

Finally we have a formula to calculate an individual worker thread's service rate. From now on we call this new queuing system as M/M/1/MT (Multi-Thread).

### 3.2. Architecture and Operation Steps of Autonomous Delay Regulation for Multi-Threaded Internet Servers

The overall architecture is shown figure 2. The M/M/1/MT Queuing Model Predictor first computes the service rate  $\mu$  for worker threads necessary to achieve a specified average delay  $D_{ref}$ , given the currently observed average request arrival rate  $\lambda$ . Server resources are then allocated to achieve the computed thread-base service rate using the sleep actuator. Then, the feedback control loop compares the actual delay  $D$  achieved to the desired average  $D_{ref}$  and adjusts the resource allocation accordingly in an incremental manner to ensure that the desired delay is maintained.

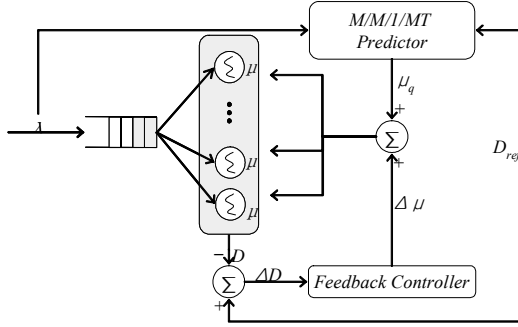


Figure 2. queuing Model-Based Feedback Control Architecture

**Step 1:** At each control invocation, we measure the average service time of threads (HTTP processing time)  $\bar{S}$  and the average number of active threads  $\bar{k}$  and update the request rate estimate  $\lambda$ .

**Step 2:** Based on results from M/M/1/MT queuing model, the queuing predictor outputs a new service rate for worker threads

$$\mu_q = \frac{1}{\min(m, k)} \left( \frac{1}{D_{ref}} + \lambda \right)$$

**Step 3:** If the measured mean delay is different from the reference delay,  $D_{ref}$ , an error is built up and a controller produces as feedback term to be added to the service rate computed by the queuing predictor. Then, the new service rate for the next interval is

$$\mu = \mu_q + \Delta \mu.$$

**Step 4:** After calculating target service rate  $\mu$  for worker threads, resource allocation is enforced to adjust the service rate to the target service rate.

### 3.3. Enforcing Resource Allocation By Sleep Actuator

After a feedback controller calculates a service rate adjustment based on the difference between the delay reference and measured delay in each control interval, the adjusted service rate is then used to control resource allocation in the server using a resource allocation module, *actuator*. There exist several techniques for allocating system resources in literature [18] [7] [11] [13] [23] [15] [29] [30]. One of the most popular ways of ad-

justing the service rate of the system is by changing the number of threads. The performance of this actuator assumes that the service rate of the system is proportional to the number of allocated worker threads. One downside of this mechanism is that the performance of the actuator is dependent on the characteristics of a specific workload and the application structure. Hence offline system profiling is typically desirable to find an approximate value for the service rate per thread allocation [9] [14][21][23].

In this section, we introduce the *sleep actuator* to solve these drawbacks. The basic idea is simple; worker threads are enforced to sleep for a certain amount of time to match the calculated service rate by the controller. This follows from observation that by changing service time of request processing, the service rate can be properly adjusted. To this end, sleep time is dynamically adjusted based on the difference between the current measured server service rate and the target service rate.

Let  $\mu(n+1)$  be the target service rate of worker threads for the next interval produced by the feedback controller.  $\mu(n)$  denotes the current service rate. Let  $X(n+1)$  be the sleep time to be applied for the next interval and  $X(n)$  be the current sleep time enforced at current interval. Sleep time for the next interval  $X(n+1)$  is obtained incrementally such that the sleep time adjustment  $\Delta X$  is calculated based on the difference between current measured service rate of worker threads  $\mu(n)$  and the target service rate  $\mu(n+1)$ . Formally, the sleep time for the next interval  $X(n+1)$  is calculated as

$$X(n+1) = X(n) + \Delta X, \quad X(0) = 0 \quad (8)$$

$$\Delta X = \frac{1}{\mu(n+1)} - \frac{1}{\mu(n)} \quad (9)$$

For example, in figure 3, if the current service  $\mu(n)$  is 1 (service time is 1) and the service rate for the next interval  $\mu(n+1)$  is 0.5 (service time is 2), then  $\Delta X$  will be 1, which leads to larger sleep time. Likewise, if the current service  $\mu(n)$  is 0.5 (service time is 2) and the service rate for the next interval  $\mu(n+1)$  is 1 (service time is 1), then  $\Delta X$  will be -1, which leads to smaller sleep time. To help response time converges faster to the set point, instead of sleeping for the entire given amount time  $X(n+1)$  set by controller, it spins around a loop in which a small amount of sleep is repeatedly enforced. This enables to change the sleep time of a request currently being serviced to the newly adjusted sleep time.

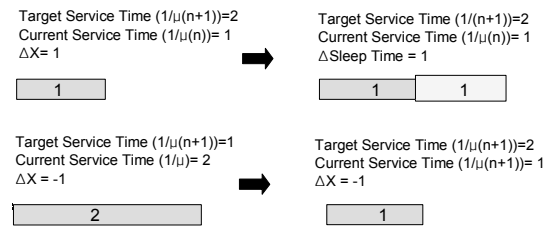


Figure 3. Conceptual Diagram of Sleep Actuator

Figure 4 (a), (b) shows the relationships of sleep time and performance metrics: CPU usage and throughput. To see the relationships more clearly, a simple workload is applied where inter-arrival time is constant reducing the effect of input fluctuation. Sleep time ranging from 0.01 second to 0.5 second is enforced for every request processed. Figure 4 (a) demonstrates the relationship of CPU usage and amount of sleep time enforced. It clearly shows that as sleep time gets larger, CPU usage decreases. This is attributed to the fact that imposing more sleep time gives worker threads less time to run, idling CPU time. Note that, after 0.2 sleep time, CPU usage does not decrease anymore, because kernel still has to handle incoming requests. Likewise, in figure 4 (b) larger sleep time results in smaller throughput and smaller sleep time yields larger throughput. From this result, it can be concluded that, by adjusting sleep time, we effectively have control of CPU, thus changing service rate properly.

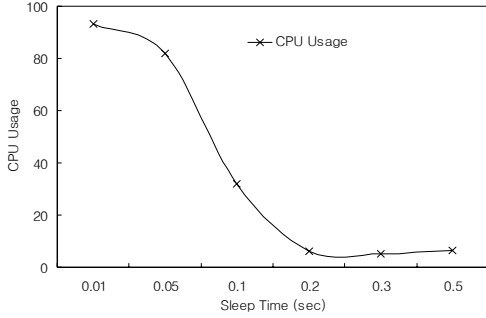


Figure 4 (a). CPU Usage and Sleep Time

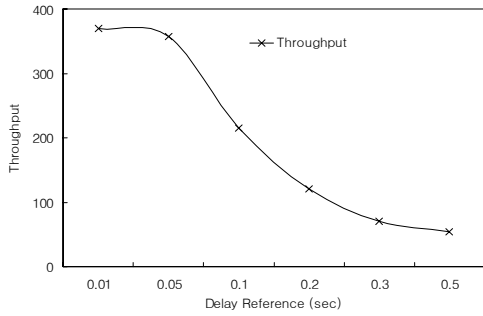


Figure 4 (b). Thoughtput and Sleep Time

### 3.4. Design of Feedback Controller

In this section, we describe a design procedure for a feedback controller for the proposed delay regulation framework. The abstraction for a multi-threaded Internet server is M/M/1/MT queuing system. We use a P (Proportional) controller for feedback control because it is a simple way to relate control error and we will show later in this paper that P controller is sufficient for correcting residual error caused by M/M/1/MT queuing model. In order to design a P controller, the desired output value should be first specified and this is called *reference signal* which is denoted by  $r(n)$  for  $n$ -th time interval.  $u(n)$  denotes control output and  $e(n)$  denotes the control error,  $r(n)-y(n)$ ,

where  $y(n)$  is system response. Then, a digital form of the P control function is,

$$u(n) = K_p e(n)$$

Intuitively, the adjustment of the tuning control depends on the current value of the control error  $e(n)$ . Its effect on the control output  $u(n)$  are weighted by the proportional control gain  $K_p$  for the response to system disturbances. In the case of queuing Model-Based Feedback Control,  $y(n)$  is an average response time of  $i$ -th interval and  $u(n)$  is to be service rate applied for worker threads.

Since we use M/M/1/MT queuing model as system model, we differentiate equation (6) to linearize at the operating point

$$\frac{\partial D}{\partial \mu} = -\frac{\min(m,k)}{(\min(m,k) \times \mu - \lambda)^2} \quad (10)$$

At the operating point, this yields

$$\frac{\partial D}{\partial \mu} = -\min(m,k) \times D_{ref}^2 \quad (11)$$

Since we are dealing with a discrete case, the above differential equation approximates to a difference equation.

$$\frac{\Delta D}{\Delta \mu} = -\min(m,k) \times D_{ref}^2 \quad (12)$$

Finally, the proportional P controller is implemented according to

$$\Delta \mu = -\frac{\alpha}{\min(m,k) \times D_{ref}^2} \times \Delta D \quad (13)$$

where  $\alpha$  is a control gain. We choose  $\alpha$  empirically such that it guarantees a reasonably fast converging time while maintaining stability upon input fluctuation. In our test implementation with Apache Web server, we use  $\alpha=2.0/D_{ref}$ . Given values of maximum number of worker threads  $m$ , delay reference  $D_{ref}$ , the current number of active worker threads  $k$ , and error  $\Delta D$ , incremental change of service rate for each worker thread  $\Delta \mu$  can be easily calculated.

## 4. Experimental Validation

We have implemented the proposed delay regulation framework for multi-threaded Internet servers using Apache web server 2.0.7 and Linux kernel 2.4.20. We choose Apache web server because it is an open source project and also one of the most popular Web servers. In our implementation, Apache is modified to provide state information used in the control system.

### 4.1. Implementation Details

Tinyproxy 1.6.1 is used and modified to implement the sensor (monitor), controller and actuator modules. Tinyproxy [24] is a lightweight and fast HTTP proxy that consumes fewer resources than fully equipped proxies. In our test-bed, all HTTP requests from clients flow through Tinyproxy and are forwarded to Apache Web server. Responses from the Web server also return back to clients through Tinyproxy. The sensor monitors both HTTP requests from clients and responses from the Web server. It calculates and estimates parameters and performance metrics such as arrival rate, average response time, average ser-

vice time, service rate of each worker thread and the number of active threads.

At each control period, above mentioned parameters are estimated and then sent to the controller module via shared memory since Tinyproxy exploits multi-process architecture. The controller is invoked to determine the next service rate. It is composed of two parts: The queuing model predictor takes the measured client request rates and the number of active worker threads from the sensor module and produces the model predicted service rate; The feedback controller takes the measured average response time as input and produces service rate adjustment. Then the actuator calculates sleep time with respect to the service rate produced by the controller to enforce proper sleep time. Sleep actuator is implemented in Apache with less than 50 lines added at most, and `usleep()` system call is used to force worker threads to sleep for a certain amount of time to adjust the processing time (service time) of requests. Sleep time is dynamically adjusted based on the difference between current measured server service rate and the target service rate (Controller's output), hence makes service rate match the target service rate.

Since HTTP processing steps can be divided into three steps and the second step (the requested objects are read from the storage or generated dynamically) the third step (the responses are replied back to the clients) can be overlapped especially when the object is a dynamic page, it is reasonable to insert the sleep actuator between the first step and the second step. Also, since a request can be rejected if the requested URL is not proper, it is not good idea to put it in front of step 1, either. As estimating parameters such as response time and arrival rate, the presence of noise can incur overreaction of the controller, hence it might impair the overall performance. Preventing this from happening, exponentially moving-average filter were applied.

## 4.2. Experimental Setup

All experiments are conducted on a test bed consisted of two PCs connected through 100Mbps Ethernet. The client machine is equipped with a 1.7GHZ Intel Pentium IV processor and 512MB RAM. `httpperf` [19] is used as synthetic generator on the client. We modified `httpperf` to generate realistic workload from real Apache access log (i.e. web traces explained below). The server machine has a 1GHZ Intel Pentium III processor and 256MB RAM, which runs Apache 2.0.7 and Linux kernel 2.4.20.

Experiments are performed using the World Cup 98 Web trace [6]. The trace consists of all the requests made to the World Cup Web site between April 30, 1998 and July 26, 1998. Each request is recorded with an arrival timestamp, a client ID for each IP address, size of the requested file and other fields.

## 4.3. Results and Model Validation

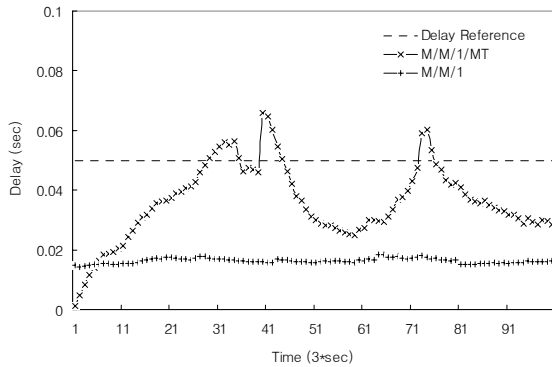
We conduct a set of experiments for the purpose of differentiating the performance of the proposed autonomous delay regulation framework from those of previous approaches. We compare the performance of four different types of controllers each other:

- M/M/1 Queuing Predictor Only
- M/M/1/MT Queuing Predictor Only
- Queuing Model-Based Feedback Controller (M/M/1+P Controller)
- The Proposed Autonomous Delay Regulation Framework for Multi-Threaded Internet Servers

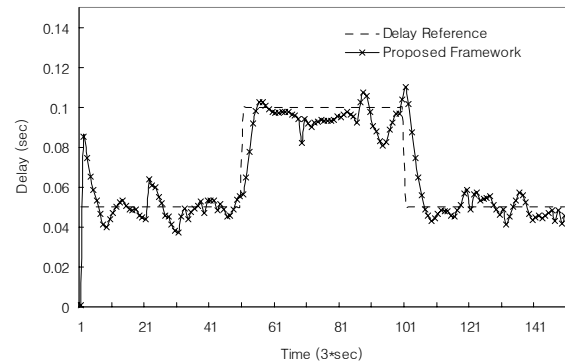
For all experiments shown here, the reference delay was set to 0.05 second and the length of control interval is 3 second. At each control period, the average request arrival rate, response time of that interval and active number of threads are measured.

First, we first compare M/M/1 queuing predictor and the M/M/1/MT queuing predictor. The purpose of this experiment is to see the performance difference of two queuing models and also, how much error is incurred due to model inaccuracy. Figure 5 shows the results. M/M/1 queuing predictor exhibits larger error with regard to the reference value. This is expected, since M/M/1 formula does not take into account the consequence of the multi-threaded architecture so that the calculated service rate is much higher than the proper value to reach the set point. In comparison, M/M/1/MT exhibits much better performance than M/M/1 model since it considers the active thread number. However, it still shows substantial error compared to the reference value. This is attributed to the two facts. First, we use the steady state result of the queuing model to calculate. Hence the controller cannot react fast enough to catch the input load change. This explains the delay fluctuation in interval (30, 45) and (70, 75). Secondly, even though M/M/1/MT includes the effect of the multi-threaded architecture, simplified assumptions such as exponential service time or the ignorance of the concurrency gain still leads to the sizable residual error.

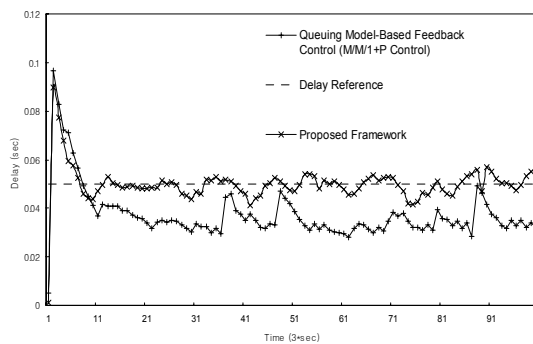
Figure 6 presents the comparison of the delay of Queuing Model-Based Feedback Control and that of the proposed approach. We used the P controller for feedback controllers in both cases. Queuing Model-Based Feedback Control reduces residual error significantly compared to M/M/1 Queuing Predictor Only, but it still shows error. This is because the P controller cannot correct the residual error since the service rate prediction from M/M/1 queuing predictor deviates from the reference severely. In contrast, the proposed approach demonstrates much better performance and follows the reference value accurately, since the M/M/1/MT predictor determines the output service rate accurately so that the P controller can suppress the error caused by model inaccuracy with no trouble. Note that with PI or PID controller, Queuing Model-Based Feedback Control can perform well [21] but parameter tuning of PI or PID controllers is not a trivial matter which requires system identification phase for that. Figure 7 shows the performance of the proposed scheme when reference delay changes during the execution. The purpose of this experiment is to see how the proposed scheme adapts well autonomously under the changing environments without intervention of administrators. Since we are already using a realistic workload from real Web traces, instead of changing a workload,



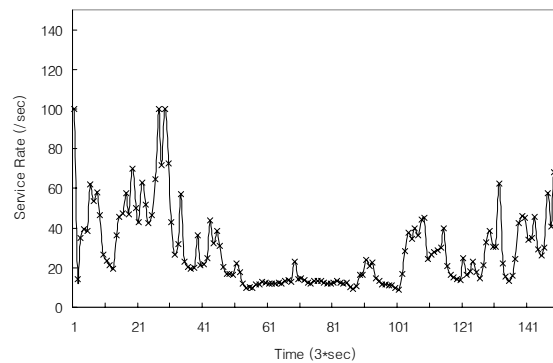
**Figure 5. Comparison between M/M/1 Queuing Predictor and M/M/1/MT Queuing Predictor**



**Figure 7. Changing Reference Delay**



**Figure 6. Comparison between Queuing Model-Based Feedback Control (M/M/1+P Control) and the Proposed Framework**



**Figure 8. Service Rate Adjustment**

we changed the reference delay to test the performance. As you see from figure 7, it tracks the reference value quickly without fluctuation. In figure 8, service rate is changed accordingly to follow the change in the reference delay.

## 5. Related Work

The role of feedback control has been applied to controlling software systems during latest years. Especially, control-theoretic approaches have been applied to server performance control. In [3], Abdelzaher et al. build a feedback control loop for Apache Web server [5] that enforces desired relative delays among different service classes via dynamic connection scheduling and process reallocation. In [15], a similar approach is used for Squid proxy server to guarantee cache hit-ratio by dynamically adjusting the disk space allocation. In [8], the parameters (i.e. KeepAlive time, MaxClients) of an Apache Web server are dynamically allocated using a MIMO feedback controller. The goal is to keep the system's CPU and memory utilization stabilized at a desired reference value. A similar approach is also used in Lotus Notes Server [9]. These approaches view the server system as a state space model and use a linear feedback control scheme without a queuing predictor. Due to the fundamental difference between a mechanical system and computing system, state space models cannot be built directly to reflect the

internal dynamics of the server. Instead, they are usually constructed offline using model identification techniques under certain predefined workloads [31]. Due to the stochastic nature of the Web traffic (with temporally and spatially variation), models thus obtained are not accurate. Furthermore, since computing systems are highly nonlinear [2], these models are at best a linear approximation of the real system. This leads to the problem of poor robustness when directly applying classical control theory to controlling server's performance, especially in presence of dynamic traffic loads. To solve this problem, Lu et al. [15] propose an adaptive control technique for Web caching systems. The introduction of adaptive controller helps adjusting the model to the traffic dynamics to some extent. However, the model and control are still intrinsically linear; and the adaptive controller usually responses slowly to abrupt traffic changes.

In [21], Sha et al. proposed Queuing Model-Based Feedback Control to achieve delay regulation in Web servers. It consists of a feed forward predictor and a feedback controller. The feed forward predictor approximates the queuing behavior of the server system and outputs a service rate. Then feedback controller calculates a service rate adjustment based on the difference between the delay reference and measured delay to reduce the error.

## 6. Conclusions and Future Work

In this paper, we present an autonomous delay regulation framework for multi-threaded Internet servers. To deal with the inaccuracy of a queuing predictor, we derive a queuing formula to capture the multi-threaded nature of Internet servers so that the queuing model predictor controls the system state near an equilibrium operation point, in spite of changes in the arrival process. Thanks to the accuracy of the new queuing model for multi-threaded Internet servers, a simple feedback loop effectively corrects residual errors. In order to allocate resource appropriately, we devise a sleep actuator which regulates the service rate by enforcing worker threads to sleep for certain amount of time. In this way, the proposed scheme works autonomously with no need for offline system identification. Evaluation is done thoroughly through experiments on an Apache web server using World Cup 98 Web trace. We demonstrate that the proposed delay regulation architecture indeed not only eliminates dependency to particular workloads or specific Internet server applications but also keeps delay to the reference value accurately under realistic workload and highly dynamic environments. These initial results are very promising and we hope this approach can be applied to the autonomous control of other Internet servers such as database systems. We also intend to implement and evaluate the proposed scheme in multiple-tier Internet service systems on the current test bed.

## 7. References

- [1] D. Menasce and V. Almeida, *Capacity Planning for Web Services: Metrics, Models, and Methods*: Prentice Hall PTR, 2001.
- [2] R. Jain, *The Art of Computer Systems Performance Analysis*: John Wiley & Sons, 1991.
- [3] T. F. Abdelzaher, C. Lu, "Modeling and Performance Control of Internet Servers", Invited Paper, 39th IEEE Conference on Decision and Control, Sydney, Australia, December 2000
- [4] G. Banga, P. Druschel, "Measuring the Capacity of a Web Server", In USENIX Symposium on Internet Technologies and Systems (USITS). Monterey, CA, Dec 1997
- [5] The Apache Software Foundation, <http://www.apache.org>
- [6] M. Arlitt and T. Jin, "1998 World Cup Web Site Access Logs", August 1998. Available at <http://www.acm.org/sigcomm/ITA/>
- [7] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A new Facility for Resource Management in Server Systems", In Third USENIX Symposium on Operating Systems Design and Implementation, New Orleans, Louisiana, February 1999
- [8] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, D. M. Tilbury, "Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics With Application to the Apache Web Serve", *Network Operations and Management*, 2002
- [9] N. Gandhi, S. Parekh, J. Hellerstein, D.M. Tilbury, "Feedback Control of a Lotus Notes Server: Modeling and Control Design", *American Control Conference*, 2001
- [10] V. Jacobson, "Congestion Avoidance and Control", *Proceedings of SIGCOMM'88*, ACM, August, 1988
- [11] M. Jones, D. Rosu, and M.-C. Rosu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", In 16th ACM Symposium on Operating Systems Principles, Saint-Malo, France, October 1997
- [12] K. Kant, Y. Won, "Server Capacity Planning for Web Traffic Workload", *IEEE transactions on knowledge and data engineering*, Oct 1999
- [13] R. Kumar, K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: A Resource-centric Approach to Real-Time systems", In *Proc. of the SPIE/ACM Conference on Multimedia Computing and Networking*, Jan. 1998
- [14] Y. Lu, A. Saxena, T. F. Abdelzaher, "Differentiated Caching Services: A Control-Theoretical Approach", *International Conference on Distributed Computing Systems*, Phoenix, Arizona, April 2001
- [15] Y. Lu, C. Lu, T. Abdelzaher, G. Tao, "An Adaptive Control Framework for QoS Guarantees and its Application to Differentiated Caching Services", *IWQoS*, Miami Beach, FL, May 2002
- [16] Y. Lu, T. Abdelzaher, C. Lu, L. Sha, X. Liu, "Feedback Control with queuing-Theoretic Prediction for Relative Delay Guarantees in Web Servers". *IEEE Real Time Technology and Applications Symposium*, Washington DC, 2003
- [17] D. Menasce, V. Almeida, "Capacity Planning for Web Services: Metrics, Models, and Methods", Prentice Hall, 2001
- [18] C. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications", In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90-99, May 1994
- [19] D. Mosberger and T. Jin, "httperf—A Tool for Measuring Web Server Performance", In *Proceedings of WISP '98*, Madison, Wisconsin, USA, June 1998
- [20] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin, "On Task Schedulability in Real-Time Control Systems", *Proceedings of Real-Time Systems Symposium*, 1996
- [21] L. Sha, X. Liu, Y. Lu, T. Abdelzaher, "Queuing Model-Based Network Server Performance Control", *IEEE Real-Time Systems Symposium*, Phoenix, Texas, December, 2002
- [22] J. Beran, "Statistics for Long-Memory Processes", Chapman & Hall, London, 1994.
- [23] H. Jamjoom, C. Chou, K. G. Shin, "The Impact of Concurrency Gains on the Analysis and Control of Multi-threaded Internet Services," in *IEEE Infocom*, Hong Kong, March 2004.
- [24] R. J. Kaes and S. Young, "tinyproxy," <http://tinyproxy.sourceforge.net/>.
- [25] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers, 2001.
- [26] H. Chen and P. Mohapatra, "Session-Based Overload Control in QoS-Aware Web Servers," presented at *IEEE INFOCOM*, 2002.
- [27] L. Cherkasova and P. Phaal, "Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites," *IEEE Trans. Comput.*, vol. 51, pp. 669–685, 2002.
- [28] J. Cao, M. Andersson, C. Nyberg, and M. Kihl, "Web server performance modeling using an M/G/1/K\*PS queue," presented at *10th International Conference on Telecommunications (ICT 2003)*, 2003.
- [29] Vivek Sharma, Arun Thomas, Tarek Abdelzaher, Kevin Skadron, "Power-aware QoS Management in Web Servers," *Real-Time Systems Symposium*, Cancun, Mexico, December 2003
- [30] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *Real-Time Systems Symposium*, London, UK, December 2002.
- [31] L. Ljung, *System Identification Toolbox*, Mathworks, April, 2001
- [32] L. Klienrock, "Queuing Systems, Volume I", Wiley Interscience, 1976.
- [33] L. Cherkasova and P. Phaal, "Session-based admission control: A mechanism for peak load management of commercial web sites," *IEEE Trans. Comput.*, vol. 51, no. 6, pp. 669–685, 2002.