June, 2013

Adaptive Control of Apache Web Server

Erik Reed, *Carnegie Mellon University*
Abe Ishihara, *Carnegie Mellon University*
Ole J Mengshoel, *Carnegie Mellon University*

Adaptive Control of Apache Web Server

Erik Reed
Carnegie Mellon University
Moffett Field, CA 94035
erikreed@cmu.edu

Abe Ishihara
Carnegie Mellon University
Moffett Field, CA 94035
abe.ishihara@sv.cmu.edu

Ole J. Mengshoel
Carnegie Mellon University
Moffett Field, CA 94035
ole.mengshoel@sv.cmu.edu

Abstract

Traffic to a Web site can vary dramatically. At the same time it is highly desirable that a Web site is reactive. To provide crisp interaction on thin clients, 150 milliseconds has been suggested as an upper bound on response time. Unfortunately, the popular Apache Web server is limited in its capabilities to be reactive under varying traffic. To address this problem, we design in this paper an adaptive controller for the Apache Web server. A modified recursive least squares algorithm is used to identify system dynamics and a minimum degree pole placement controller is implemented to adjust the maximum number of concurrent connections. Experimentally, we show that the controller effectively regulates the reply time of HTTP connection requests, and hence provides reactive response, by limiting the maximum number of connections accepted by an Apache Web server.

1 Background

Abrupt spikes in the usage of certain phrases ("lipstick on a pig," "our entire economy is in danger," "who is the real Barack Obama") in the daily news cycle have been observed [14]. Corresponding to such spikes, as well as other time-varying phenomena, traffic to a Web site can vary dramatically. In spite of such varying traffic, it is highly desirable that a Web site is reactive. To provide a crisp interactive experience on thin Web clients, 150 milliseconds has been recommended as an upper bound on response time [22]. In fact, quick and reasonably predictable user response is essential in all computer systems, not only Web services.

One way to ensure quick and predictable user response is through load balancing [7]. A single server's ability to service clients is bound by multiple factors, including CPU utilization, memory capacity, bandwidth capacity, and I/O rate. Load balancing at the cluster level allows connections to be distributed to servers with the least load. Typical implementations of load balancing deal with round robin balancing for domain name service (DNS) [7]. Much of existing frameworks and previous research deal with inter-server connections and balancing, with little or no focus on dynamic parameter adjustment of individual servers.

Focusing on an individual Web server rather than a cluster of machines, we investigate in this paper an alternative approach to ensuring quick and predictable user response, namely feedback control and specifically Minimum Degree Pole Placement control [5]. This can be in an environment where inter-server load balancing is already in place, or when a single server is handling Web traffic. Our interest is in the ability to manipulate parameters online to control server load in order to maximize the efficiency of each server individually under varied and changing conditions. This paper uses the Apache Web server,[1] a popular open source package [13].

Apache consists of a structured pool of workers, each its own process.[2] A master process, the Apache daemon, listens for requests and delegates HTTP communications to the worker processes. To control the rate of incoming connections, an Apache server administrator can adjust two primary settings: *MaxClients* (MC) and *KeepAlive* (KA). Depending on server load and Web activity, the Apache daemon maintains up to MC worker processes, which can in one of three states at any time: busy, idle, or thinking. Busy workers are currently processing a request and awaiting a reply; thinking workers are waiting for an HTTP request after an established connection; and idle workers are waiting for a new connection request. The KA parameter determines how long workers maintain their current connection with a client before terminating the connection (*i.e.* changing their state from thinking to idle). Apache 2.2 has default settings of $\texttt{MC} = 150$ worker processes and $\texttt{KA} = 5$ seconds.

[1] http://httpd.apache.org/

[2] This is on a Unix-based system. Apache on Windows, which we do not investigate here, is different.

This paper uses performance from the user's perspective as a metric and input to the controller, rather than CPU- and memory-based metrics. In contrast to previous research (Section 2), our experiments include metrics such as time to connect, reply time, and requests per second. With these in mind, we derive a model in real-time and adaptive controller (Section 3). We then perform preliminary server benchmarking, an open loop simulation, and lastly test the controller using mean HTTP reply time as a setpoint (Section 4). While we use reply time as the output modeled by the system, this approach can be applied using any other metric collected by the server, such as throughput, connection error frequency, or mean time until a connection is established.

2 Related Work

Apache Web server load has previously been modeled as a linear multiple-input multiple-output (MIMO) system [8], using input parameters KA and MC and output parameters memory and CPU usage. This MIMO model expands on previous work of single-input single-output (SISO) controllers in computing systems – in particular for congestion control [10] and controlling Web server delay [16]. Robertsson *et al*. used a PI controller and nonlinear model with simulations using Matlab [20]. Abdelzaher *et al*. explored average requests delay as a setpoint using a feedforward predictor [1–3].

Another SISO controller using KA and MC as parameters has been researched [11]. Notably, the authors concluded that SISO is insufficient to obtain accurate models for CPU, as KA and MC are not independent. The use of CPU, memory, KA , and MC was further investigated by Hellerstein and Diao, along with a queueing model to predict server response time [9, 12]. They proposed a MIMO implementation able to capture the interactions of KA and MC for modeling memory and CPU usage [11]. To control KA and MC, multiple SISO controllers (with separate controls loops for both KA and MC) were very effective when compared to a single MIMO controller. Also of interest is their averaging of data points over time before performing controller calculations.

A simple, first order linear time invariant MIMO model was created [8] using inputs KA, MC and outputs CPU, MEM:

$$\begin{bmatrix}\mathrm{CPU}_{k+1}\\ \mathrm{MEM}_{k+1}\end{bmatrix} = \mathbf{A}\begin{bmatrix}\mathrm{CPU}_{k}\\ \mathrm{MEM}_{k}\end{bmatrix} + \mathbf{B}\begin{bmatrix}\mathrm{KA}_{k}\\ \mathrm{MC}_{k}\end{bmatrix}, \qquad (1)$$

where $\mathbf{A},\mathbf{B} \in \mathbb{R}^{2\times 2}$ were estimated via least squares regression and k represents a discrete time interval.

Various methods of traffic generation for testing performance of Web servers have also been previously researched. A closed source traffic generator termed WAGON (Web Traffic Generator and Benchmark) [15] was used to simulate user generated traffic and varying amounts of I/O, memory, and CPU intensive traffic [8]. Open source Web traffic generators include *Apache Benchmark* and *Httperf*. Httperf, which we adopt, has been shown to be effective in measuring Web server performance as well as mimicking user behavior [18].

3 Algorithms and Analysis

The underlying control approach taken is the minimum degree pole placement (MDPP) control design proposed by Astrom [5]. In this approach, feed-forward and feedback controllers are designed to force the open-loop system to follow a reference model. In the ideal case, model following is achieved perfectly. However, due to parametric uncertainty, this is not achieved in practice. This is the motivation to introduce reconfiguration to the control problem. During a change in plant dynamics, the control effectiveness changes, requiring a larger or smaller actuator signal to maintain desired performance.

3.1 Plant Modeling

There are several approaches to the modeling of computation for control applications. Here, we use linear Auto-Regressive modeling with eXogenous input or linear ARX. Nonlinear approaches, such as discrete time neural networks, may also be used. Nonlinear modeling is more complex yet may be able to capture inherent nonlinear behavior otherwise unaccounted for in ARX modeling. On the other hand, linear modeling is generally simpler to understand and implement.

We will assume a finite class of plants $P^{(i)}$ for $i \in [1,M]$; the integer i denotes a specific plant or device, such as a particular server type or configuration. Suppose the plant is described by an ARX model with an additive noise term. For simplicity, we assume Gaussian noise. Denote $P^{(i)}$, $y^{(i)}(k)$, $u(k)$, and $\eta(k)$, the plant operator, scalar output, input, and noise at sample time k, respectively.

The relationship between these quantities is given by:

$$y^{(i)}(k) = P^{(i)}u(k) = \phi^T(k-d)\theta^{(i)} + \eta(k) \qquad (2)$$

where $\phi^T(k-d)$ denotes the regression vector and consists of a tapped delay line of input and output measurements, and $\theta^{(i)}$ denotes a vector of parameters corresponding to the ith plant. A number of methods exist to estimate $\theta^{(i)}$ in both batch and on-line modes. Similar ARX models have been used in modeling a number of digital processes [11]. In the following we will drop the superscript (i) from the plant parameter vector. The purpose of the superscript was to indicate that there exists

a family of (unknown) parameter vectors that adequately model various plant scenarios described above. The purpose of the system identification algorithm described below is to track and identify these parameter vectors that may change gradually or abruptly during online control.

The reconfigurable control approach estimates θ online and then uses the estimates to update a control law (see Section 3.2) below. To estimate θ, it is typical to use the recursive least squares (RLS) method, in which we minimize the cost function

$$J = \sum_{k=1}^{N} \lambda^{N-k} e^2(k), \tag{3}$$

where $e(k)$ is the error between the true and estimated outputs, and $\lambda \in (0,1]$ is the forgetting factor.

The RLS method can lead to two problems when attempting to track varying parameters. First, a small forgetting factor, needed to track fast or abrupt parameter variations can cause a large covariance matrix which could lead to covariance "blow up". Second, as the forgetting factor is decreased, the size of the data window gets smaller and it is more likely that there will exist data collinearities within the data window. The modified sequential least squares (MSLS) algorithm, which prevents singularities in the covariance matrix [17], has been proposed to deal with this issue. This is due to the reformulation of the least squares problem. The cost function to be minimized includes additional penalties on changes in the parameter values in the form of temporal and spatial constraints. Weighting matrices are included to adjust the extent of penalization on each parameter variation. For this case, we will define the cost function as

$$J = \sum_{k=1}^{t} \|y(k) - x^T(k)\theta(t)\|^2 \lambda^{t-k} + m\|\theta(t) - \theta(t-1)\|^2$$

where $\theta(t)$ corresponds to the parameter estimates at time t.[3] Setting $\frac{dJ}{d\theta} = 0$, we get the solution

$$\theta(t) = \left[\mathscr{X}^T \tilde{\mathscr{X}} + mI\right]^{-1} \left[m\theta(t-1) + \tilde{\mathscr{X}}^T y(t)\right],$$

where

$$\begin{aligned} \mathscr{X}^T &= \begin{bmatrix} x(1) & x(2) & \dots & x(t) \end{bmatrix} \\ \tilde{\mathscr{X}}^T &= \begin{bmatrix} x(1)\lambda^{t-1} & x(2)\lambda^{t-2} & \dots & x(t) \end{bmatrix} \\ y^T(t) &= \begin{bmatrix} y(1) & y(2) & \dots & y(t) \end{bmatrix}. \end{aligned}$$

The covariance matrix in this algorithm is now

$$P(t) = \left[\mathscr{X}^T \tilde{\mathscr{X}} + mI\right]^{-1}.$$

A recursive version, derived by Bodson [6], is given as

$$\begin{aligned} \theta(t+1) = \theta(t) + P(t+1)x(t+1)\left[y(t+1) - x^T(t+1)\theta(t)\right] \\ + m\lambda P(t+1)\left(\theta(t) - \theta(t-1)\right). \end{aligned}$$

In conventional RLS with forgetting factor, it is typical to use the Matrix Inversion Lemma (MIL) to calculate $P(t+1)$. However, due to the additional penalty term, the MIL leads to the inversion of a $(1+m)\times(1+m)$ matrix, where m is the number of unknown parameters. Thus, the MIL does not help in this case, and hence, we opt to directly take the inverse of $P(t+1)$.[4]

Upon a system or configuration change we would like to adapt our controller when the parameter estimates converge. The convergence time may vary depending on factors such as input excitation. We decide to monitor the posterior prediction error

$$\text{Prediction Error} = \left|y(t) - x^T(t)\theta(t-1)\right|. \tag{4}$$

A tolerance is defined, and we adapt our controller when

$$\text{Prediction Error} > \text{Tolerance}.$$

3.2 Minimum Degree Pole Placement

We base our minimum degree pole placement (MDPP) controller on previous work [5]. For MDPP, let $H(q)$ be the transfer function of the plant, where q is the forward shift operator, and $A_m(q), B_m(q)$ be the reference model denominator and numerator respectively:

$$H(q) = \frac{b_0 q + b_1}{q^2 + a_1 q + a_2}.$$

The steady state is given by:

$$\frac{B_m(1)}{A_m(1)} = \frac{B'_m(b_0+b_1)}{1+a_{m_1}+a_{m_2}} = 1$$

$$B'_m = \frac{1+a_{m_1}+a_{m_2}}{b_0+b_1},$$

where A_{m_1} and A_{m_2} are chosen parameters. Denote A, R, S, T and $B = B^+B^-$ to be polynomials in q where B^+ is a monic polynomial with stable zeros and B^- corresponds to the unstable factors. The Diophantine equation, or closed-loop characteristic polynomial, is defined by

$$AR' + B^-S = A_0 A_m.$$

Let $R' = \frac{R}{B^+} = R$. Then,

$$(q^2 + a_1 q + a_2)R' + (b_0 q + b_1)S = A_0(q^2 + A_{m_1} q + A_{m_2}).$$

[3] Here, t replaces N in (3) to indicate that there is a moving window associated with the parameter estimates. The control system at time t uses the latest parameter estimates given by $\theta(t)$.

[4] For higher order systems, this approach may not be acceptable. Ways to reduce the order of this computation have been proposed [6].

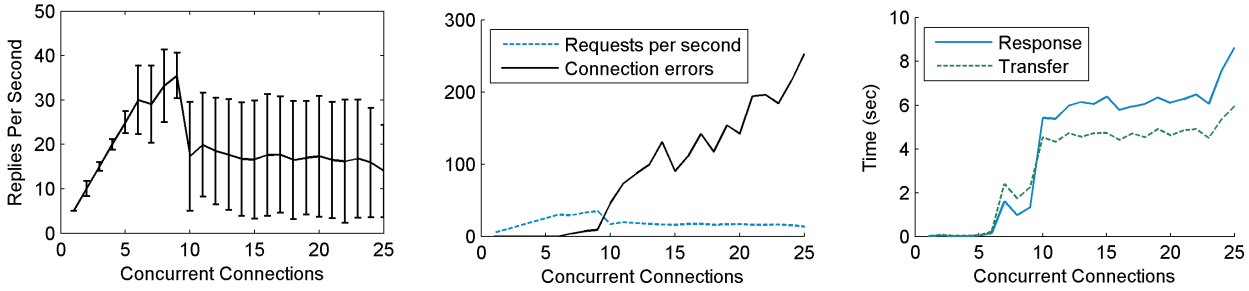


Figure 2: The effect of increasing the number of connections, for the Apache Web server, on its performance. (a) shows the RPS increasing until 9 concurrent connections, at which point the standard deviation increases and mean RPS decreases. The connection errors are shown in (b) and steadily rise after 9 concurrent connections. The effect of concurrent connections on mean HTTP request response (T_R) and transfer (T_T) times are shown in (c).

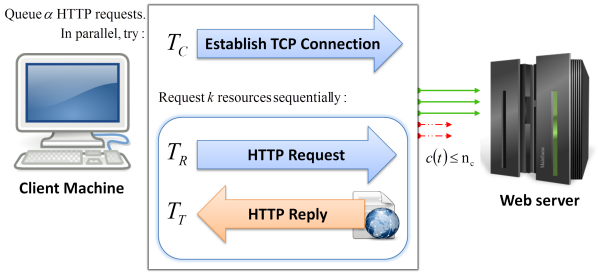


Figure 1: The procedure for generating load and limiting the number of concurrent connections $c(t)$. In this example, $\alpha = 5$ and $n_c = c(t) = 3$.

Deriving our controller u, we have:

$$R' = q + r_1 = R \tag{5}$$

$$S = s_0 q + s_1 \tag{6}$$
$$A_0 = q + a_0$$

$$(q^2 + a_1 q + a_2)(q + r_1) + (b_0 q + b_1)(s_0 q + s_1) = A_0(q^2 + a_{m_1} q + a_{m_2})$$

Lastly, we can compute T:

$$T = A_0 B'_m = (q + a_0) B'_m \tag{7}$$

Using Eq. 5, 6, and 7 to calculate R, T, we can now compute: $u = \frac{T}{R} u_c - \frac{S}{R} y$, the control signal from our control law. A more thorough derivation of Eq. 7 and this specific controller with equivalent zero cancellation is given by Reed *et al.* [19].

4 Experiments

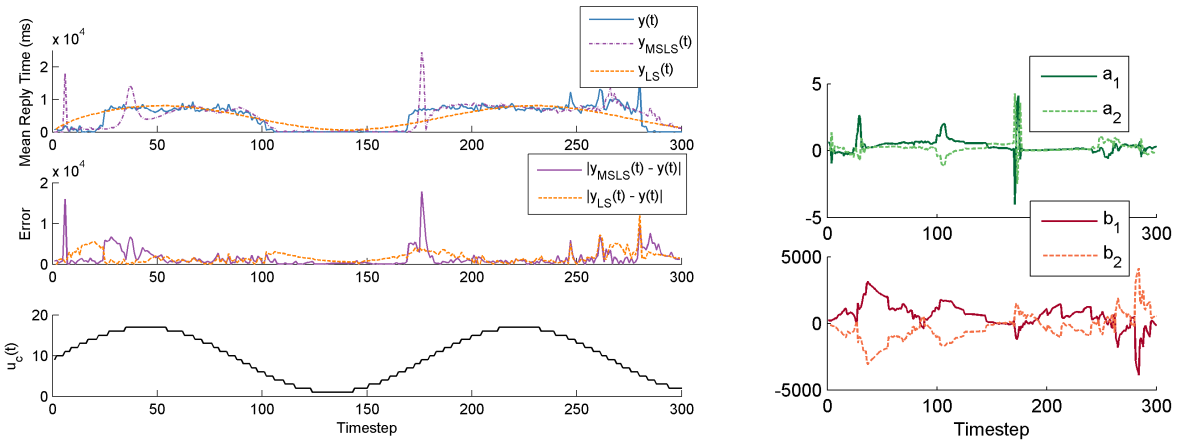
These experiments will measure the Apache Web server under varying numbers of connections, introduce an ARX model, and test an adaptive MDPP controller for setting the maximum number of concurrent connections.

4.1 Simulation Setup and Procedure

In our simulations, we used Apache v2.2 (the latest stable release at the time of writing) running on a Linux kernel 3.0.0-14 x64 workstation with a dual core Intel T2400 and 2GB of RAM. This workstation acted as the *server* and hosted PHP code simulating an active blog. The Apache Web server parameters were kept at their default settings. A second workstation of similar hardware specifications, the *client*, sent HTTP requests concurrently to the server to simulate Web traffic. The client resided on the same LAN as the server.

Figure 1 shows the interactions between the client and the server. For a single HTTP request to the server (e.g. fetching `index.html`), we collected the following statistics: time until initial TCP connection established (T_c), time until request response (T_R), time for the request transfer/reply (T_T), as well as several other metrics not shown here due to space. To generate HTTP traffic and simulate a simple user interaction with the client, we used *Httperf*.

The client proceeded to generate Web traffic by queueing a list of α TCP connection requests to the server. After a connection was established, the client sent k HTTP requests to a random HTML, Javascript, or stylesheet (CSS) resource, each with probability $\frac{1}{3}$. A single HTTP request typically resulted in a 80KB reply by the server (80KB being the mean size of the three potentially requested resources). Since a single connection cannot handle concurrent HTTP requests, the k HTTP requests were performed sequentially; immediately after a resource was received by the client, the next resource was requested. After the connection's k requests completed, the connection was terminated by the client. Up to α connections were created concurrently by the client. Requesting multiple resources per connection leverages a persistent TCP connection depending on the request frequency and `KA` setting.



(a) (b)

Figure 3: (a) Model tracking of $y(t)$ for least squares (LS) and MSLS (top), model error (middle), and the open-loop, sinusoid input for max number of connections $u_c(t)$ (bottom). (b) Learned parameter values over time for the MSLS model during an open-loop simulation (see Eq. 8).

On the server side, we denote $c(t)$ to be the number of concurrent connections at time t and n_c to be the maximum number of concurrent connections set by the server. When n_c connections are active in the server, all additional connection requests were ignored until $c(t)$ decreases. In this case, the client waited indefinitely until the server had capacity, i.e. $c(t) < n_c$ (rather than terminate the connection request after a certain amount of time). Once a connection was established, a connection failure could occur if there was a TCP socket timeout (a constant supplied by the operating system), which was 20 seconds for our version of Linux.

We approximate the output $y(t)$, the total time for an HTTP reply by the Web server ($T_R + T_T$), with $\hat{y}(t)$. To model the server, we used a 2nd order SISO model defined by

$$\hat{y}(t) = -a_1\hat{y}(t-2) - a_2\hat{y}(t-1) + b_1u(t-2) + b_2u(t-1). \quad (8)$$

The parameter tuned by the controller, $u(t)$, is the maximum number of concurrent connections (n_c). The parameters a_1, a_2, b_1, b_2 were learned offline via least squares and online via MSLS (Section 4.3, 4.4). As with previous work [8], we define $t = 10$ (seconds) as an interval during which data is collected. We used the mean reply time during this interval to measure $y(t)$.

4.2 Varying Concurrent Connections

Here we test the effect of number of concurrent connections on various metrics. On the client we set $\alpha = 1000$ and $k = 5$; on the server $n_c \in [1, 25]$. First we note that increasing the number of concurrent connections was beneficial to the server's throughput, measured by replies per second (RPS), up until a certain point (Figure 2a). There was a sharp drop in RPS at $n_c = 10$, while $n_c = 9$ yielded peak RPS. Additionally, the standard deviation of the measurements increased substantially for $n_c \geq 10$. The increase in standard deviation is a result of the server hardware limits being met; the server was unable to effectively address all the HTTP requests, resulting in connection drops or timeouts.

Figure 2b shows successful connection *requests* per second and connection errors. Connection requests per second is different from RPS in that replies are made when the connection has already been established, versus connection requests which start new TCP connections (i.e. no HTTP packet data has been sent yet). A value of $n_c = 9$ also resulted in the peak number of successful connection requests per second (30). Since 30 connections were established per second, this means that many connections were able to complete their $k = 5$ HTTP requests in less than a second.

Next, Figure 2c shows the HTTP reply response (T_R) and transfer (T_T) times. Both response and transfer times increased as the number of concurrent connections increased. At $n_c = 7$, the mean time to complete an HTTP reply ($T_c + T_T$) was larger than one second. Interestingly, as seen in Figure 2a, the RPS is still increasing at $n_c = 7$.

Using these measurements, we envision two scenario types that a server administrator may optimize for using a model and controller:

- Maximizing overall throughput (RPS) when quick HTTP replies are not necessary (such as when downloading large files or executing analytics Javascript).
- Ensuring that HTTP reply time is low enough to keep the user engaged; 150ms has been suggested as an upper bound for crisp user interaction [22].

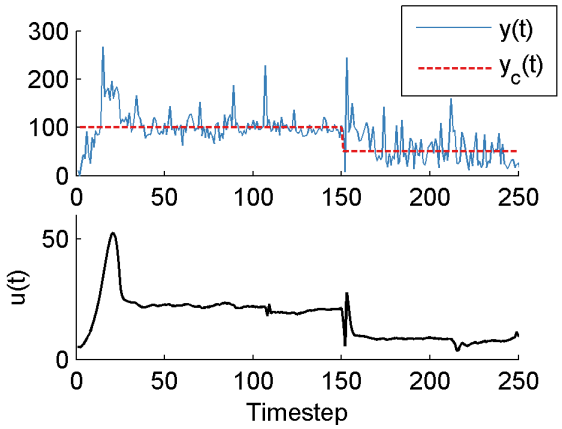


Figure 4: Setpoint tracking with a change of reply time $y_c(t) = 100$ to $y_c(t) = 50$ at timestep 150 (top) and MDPP controller value for max connections $u(t)$ (bottom). Note that $y_c(t)$ is shown in milliseconds.

In Section 4.4, we recreate the latter scenario by setting a target reply time and controlling n_c.

4.3 System Identification

In this section we perform an open-loop simulation with least squares (LS) and MSLS models. To sufficiently excite the server response, we use a sinusoidal input $u_c(t) \in [1,17]$ for the max number of connections (n_c). We set $\alpha = 1000$ and $k = 15$. After the simulation, we train the model parameters a_1, a_2, b_1, b_2 of Eq. 8 using LS and MSLS (see Figure 3a). For MSLS, we use a forgetting factor of $\lambda = .92$ and a noise reducing coefficient of $m = .2$. Recall that $y(t)$ is the mean reply time over a time interval of 10 seconds. We denote y_{LS} to be the LS estimator and y_{MSLS} to be the MSLS estimator. Note that we use batch LS rather than RLS; that is, the parameter learning is done after all the data is known and the parameters are constant through the simulation. In contrast, the MSLS parameters are learned online and are adaptive.

4.4 Adaptive MDPP Controller

This section describes two closed-loop simulations using an MSLS model for tracking $y(t)$ and MDPP control of $u(t)$. We denote $y_c(t)$ to the be setpoint, or target $y(t)$ in which the controller tunes $u(t)$ to minimize the error function $e(t) = |y_{MSLS}(t) - y_c(t)|$ (see Eq. 8 for $\hat{y}(t)$). As with Section 4.3, each timestep is 10 seconds and model parameters were set to $\lambda = .92$ and $m = .2$. We used load generation parameters $\alpha = 100$ and $k = 15$.

The first simulation used a setpoint of $y_c(t) = 100$ for $t < 150$ and $y_c(t) = 50$ for $t \geq 150$ (see Figure 4). The measurements of $y(t)$ are noisy, increasingly so when the setpoint is changed. At $t = 150$, there is an immediate shift in $u(t)$, then the model parameters begin to relax. The volatility of the a_1, a_2 parameters is a consequence of noise; this could likely be reduced with a greater forgetting factor λ and a greater m.

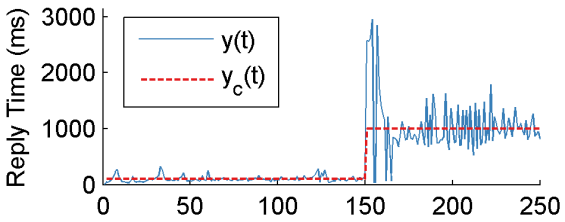


Figure 5: Setpoint tracking with a simultaneous setpoint change of $y_c(t) = 100$ to $y_c(t) = 1000$ and load generation parameter change of $\alpha = 100$ to $\alpha = 200$ at $t = 150$.

Next we performed a simulation in a high noise environment by raising the setpoint $y_c(t) = 1000$ at timestep 150. For $t < 150$, $y_c(t)$ was again kept at 100. We anticipated noise based off of our findings in Section 4.2. Furthermore, we adjust α to be 200 instead of 100 at $t \geq 150$, doubling the load in order to simulate a spike in user activity (i.e. a change in the plant dynamics). The load parameter α was also adjusted to increase the number of effects the model parameters must adapt for. This simulation is shown in Figure 5. There was an immediate spike in $y(t)$ at $t = 150$ as the controller adjusted $u(t)$ to compensate (overestimating $u(t)$). The parameters took 10 timesteps to converge to a stable set of values, even with the increased $y(t)$ noise. The parameter values and $u(t)$ are not shown here due to space.

5 Conclusion

We have shown a Web server can be modeled and controlled to enforce metrics that affect the user experience of a client machine (*e.g.* HTTP reply time). Model parameters were learned real-time, and adaptive performance was strong and ostensibly resilient to noise.

In future work, we would like to include additional metrics in both the model and the controller. There are many parameters to adjust within the Apache Web server, including those in Apache modules (like amount of compression, encryption levels, caching, etc) [4]. Additionally, metrics like disk I/O (which can cause significant latency in sites with large amounts of data [21]) and bandwidth may be useful in creating a more effective model.

References

[1] ABDELZAHER, T. F., AND BHATTI, N. Web server QoS management by adaptive content delivery. In *Proc. Seventh International Workshop on Quality of Service* (1999), pp. 216–225.

[2] ABDELZAHER, T. F., LU, Y., ZHANG, R., AND HENRIKSSON, D. Practical application of control theory to Web services. In *Proc. of the 2004 American Control Conference* (2004), vol. 3, pp. 1992–1997.

[3] ABDELZAHER, T. F., STANKOVIC, J. A., LU, C., ZHANG, R., AND LU, Y. Feedback performance control in software services. *Control Systems 23*, 3 (2003), 74–90.

[4] ABHARI, A., SERBINSKI, A., AND GUSIC, M. Improving the performance of Apache Web server. In *Proc. of the 2007 spring simulaiton multiconference-Volume 1* (2007), Society for Computer Simulation International, pp. 166–169.

[5] ASTROM, K. J., AND WITTENMARK, B. *Adaptive control*. Addison-Wesley, 1994.

[6] BODSON, M. An adaptive algorithm with information-dependent data forgetting. In *Proc. of the 1995 American Control Conference* (1995), vol. 5, pp. 3485–3489.

[7] CARDELLINI, V., COLAJANNI, M., AND YU, P. S. Dynamic load balancing on web-server systems. *Internet Computing 3*, 3 (1999), 28–39.

[8] DIAO, Y., GANDHI, N., HELLERSTEIN, J. L., PAREKH, S., AND TILBURY, D. M. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server. In *Proc. of IEEE/IFIP Network Operations and Management Symposium* (2002), pp. 219–234.

[9] DIAO, Y., HELLERSTEIN, J. L., AND PAREKH, S. Optimizing quality of service using fuzzy control. In *Management Technologies for E-Commerce and E-Business Applications*. 2002, pp. 42–53.

[10] FLOYD, S., AND JACOBSON, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking 1*, 4 (1993), 397–413.

[11] HELLERSTEIN, J., DIAO, Y., PAREKH, S., AND TILBURY, D. M. *Feedback control of computing systems*. Wiley, 2004.

[12] HELLERSTEIN, J. L., DIAO, Y., PAREKH, S., AND TILBURY, D. Control engineering for computing systems - industry experience and research challenges. *Control Systems 25*, 6 (2005), 56–68.

[13] HU, Y., NANDA, A., AND YANG, Q. Measurement, analysis and performance improvement of the Apache Web server. In *Proc. of IEEE International Performance, Computing and Communications Conference* (1999), pp. 261–267.

[14] LESKOVEC, J., BACKSTROM, L., AND KLEINBERG, J. Meme-tracking and the dynamics of the news cycle. In *Proc. of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD-09)* (2009), pp. 497–506.

[15] LIU, Z., NICLAUSSE, N., AND JALPA-VILLANUEVA, C. Traffic model and performance evaluation of web servers. *Performance Evaluation 46*, 2 (2001), 77–100.

[16] LU, C., ABDELZABER, T. F., STANKOVIC, J., AND SON, S. H. A feedback control approach for guaranteeing relative delays in Web servers. In *Proc. of Seventh IEEE Real-Time Technology and Applications Symposium* (2001), pp. 51–62.

[17] MONACO, J., WARD, D., BARRON, R., AND BIRD, R. Implementation and flight test assessment of an adaptive, reconfigurable flight control system. In *Proc. of the AIAA Guidance Navigation and Control Conference* (1997), vol. 97, p. 3738.

[18] MOSBERGER, D., AND JIN, T. Httperf – a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review 26*, 3 (1998), 31–37.

[19] REED, E., ISHIHARA, A., AND MENGSHOEL, O. J. Adaptive control of Bayesian network computation. In *Proceedings of the International Symposium of Resilient Control Systems* (2012).

[20] ROBERTSSON, A., WITTENMARK, B., KIHL, M., AND ANDERSSON, M. Design and evaluation of load control in Web server systems. In *Proc. of the 2004 American Control Conference* (2004), vol. 3, pp. 1980–1985.

[21] RUAN, Y., AND PAI, V. Understanding and addressing blocking-induced network server latency. In *Proc. of the USENIX Annual Technical Conference* (2006), pp. 143–156.

[22] TOLIA, N., ANDERSEN, D. G., AND SATYANARAYANAN, M. Quantifying interactive user experience on thin clients. *IEEE Computer 39*, 3 (2006), 46–52.