RC23437 (W0411-156) November 17, 2004
Computer Science

IBM Research Report

Control of Weighted Fair Queueing: Modeling, Implementation, and Experiences

Ronghua Zhang*, Sujay Parekh, Yixin Diao, Maheswaran Surendra
Tarek Abdelzaher*, John Stankovic*
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*University of Virginia

Control of Weighted Fair Queueing: Modeling, Implementation, and Experiences

Ronghua Zhang
University of Virginia
rz5b@cs.virginia.edu

Sujay Parekh
IBM
sujay@us.ibm.com

Yixin Diao
IBM
diao@us.ibm.com

Maheswaran Surendra
IBM
suren@us.ibm.com

Tarek Abdelzaher
University of Virginia
zaher@cs.virginia.edu

John Stankovic
University of Virginia
stankovic@cs.virginia.edu

Abstract

Feedback control of QoS-aware servers has recently gained much popularity due to its robustness in the face of uncertainty and modeling errors. Performance of servers is characterized by the behavior of queues, which constitute the main elements of the control loop. The central role of queues in the loop motivates understanding their behavior in the context of feedback control schemes. A popular queueing policy in servers where different traffic classes must be allocated a different share of a common resource is weighted fair queueing (WFQ). This paper investigates the interactions between a weighted fair queueing (WFQ) element and a feedback controller. It is shown that the WFQ element introduces challenges that render simple feedback control ineffective and potentially unstable. These challenges are systematically exposed, explained, and resolved. An extended feedback control scheme for the WFQ element is subsequently developed. The scheme is tested on an experimental prototype demonstrating higher predictability and an order of magnitude improvement in responsiveness over the initial design. The results of the paper apply in general to most systems that use a dynamic processor sharing approach for service differentiation.

1 Introduction

Feedback control theory has recently gained much popularity as the underlying analytic foundation for many performance control schemes in computing servers. The success of the control-theoretic approach is attributed to the inherent robustness of simple feedback controllers with respect to modeling errors and large fluctuations in load and resource capacity.

The fundamental reason control theory is applicable in the domain of computing servers is that server queues can generally be modeled by difference equations and hence are amenable to control-theoretic analysis. The performance of servers depends on the flow of requests through these queues as they pass through a series of stages or tiers at which they are queued for service. Feedback controllers can be designed that achieve desired end-to-end performance by manipulating the behavior of individual queueing elements. It is key to understand the interplay between the feedback controller and the queueing element in order to achieve desired guarantees. The collection of results on queue control may lead to the development of a fundamental theory for performance control in computing systems.

At present, such a theory is far from complete. The only queueing element that received much attention in the context of feedback control loops has been the FIFO queue. For example, the authors of [22], [16] used a FIFO queueing model to predict queueing performance in the context of a feedback control loop. QoS-aware servers, however, are beginning to export more sophisticated types of queues, such as priority queues and WFQ elements.

Analyzing the behavior of WFQ elements is particularly important. This is because most multi-class feedback control schemes proposed for computing servers to date achieve performance guarantees by logically allocating a separate fraction of the server's bottleneck resource to each traffic class. The extent of each share is controlled to provide the desired performance. Unfortunately, unlike disk space and memory, many resources (such as a disk head, a communication channel, or a CPU) cannot be physically partitioned. Instead, the abstraction of resource shares has to be enforced by an appropriate scheduling policy that allows different applications to believe they own a different fraction of resource bandwidth. Among the most common scheduling policies that achieve this end are those based on virtual clocks such as variants of WFQ. These policies seek to approximate Generalized Processor Sharing (GPS) and thus the resource shares allocated to different classes are determined by assigning appropriate weights to each class.

The behavior of WFQ and similar disciplines in terms of providing service differentiation and delay and jitter guarantees has been analyzed [5], [19], [3], along with flow control strategies in networks of such WFQ elements [13], [2]. In order to use WFQ for end-user QoS, however, the user-level QoS requirement must be used to correctly set the per-class weights as used by the WFQ scheduler. The effects of closing a feedback loop around WFQ queues for such QoS has not been systematically explored.

This paper is the first to systematically analyze the effects of closing a control loop around a WFQ element for setting the per-class weights. We show that understanding the behavior of this category of schedulers is crucial to allow feedback control loop designers to properly acount for the side effects introduced by virtual clocks. The performance of a simple feedback loop is shown to be very poor and often unstable. The sources of performance degradation and instability are analyzed, understood, and methodically eliminated leading to a general recipe for feedback control loop design of WFQ elements. The extended design is tested on a real Web application, demonstrating an order of magnitude imporvement in loop responsiveness over the baseline solution.

While our experimental studies are in the context of multi-class Web traffic, we believe that the issues considered are more generally applicable for any usage of WFQ where the weights must be changed dynamically. Based on this study, we propose that more feedback-driven adaptive algorithms for computing systems could be analyzed using control theory in order to improve their transient performance.

The rest of the paper is organized as follows. In Section 2, we briefly describe a straightforward feedback control solution of a WFQ scheme, based on an adaptive accept queue controller used in [21]. In Section 3, we examine the factors affecting the transient behavior of the WFQ element in the presence of the feedback controller, identify control challenges and design pitfalls, and develop a sound methodology for feedback control of WFQ systems. Remaining issues and future work are discussed in Section 4. The related work is summarized in Section 5. The paper concludes with

Section 6.

2 A Web Application

The running example of the WFQ system in this paper is a self-managing web server in which multiple classes of clients must receive differentiated service. As shown in Figure 1, incoming connection requests (TCP SYN packets) are first classified by a SYN classifier. The classifier uses the IP address and port number to classify the incoming connection requests into different service classes based on rules. Once the three-way handshake is complete, the connection is moved from the SYN queue to the accept queue of the listening socket. Rather than a single FIFO queue shared by all classes of clients as in a normal Linux kernel, a separate accept queue for each class is maintained. Requests are subsequently dequeued by threads, which in turn get enqueued for the CPU. Depending on the nature of such requests, either the TCP accept queues or the CPU could become the bottleneck. In this study, we focus on the accept queue bottleneck, which is often seen with a large number of HTTP requests using the HTTP/1.1 persistent connection option [15]. We expect that our main observations and results regarding control of WFQ elements should hold for the CPU bottleneck as well.

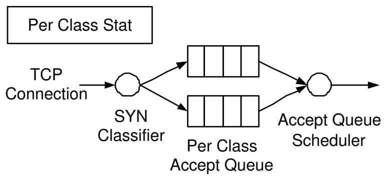


Figure 1: Kernel Enhancement

We introduce a WFQ element in the system to control the rate of accepting requests of each class (i.e., the relative dequeue rates from per-class accept queues). The vanilla Linux kernel is modified such that each class is assigned a weight, which decides the rate of accepting requests of that class. The question addressed in this paper is how to assign those weights efficiently using a feedback control scheme to achieve a stable desired per-class delay in the face of varying input load.

The algorithm used by the WFQ scheduler is start-time fair queueing (SFQ) [8]. Basically, each class is assigned a weight. When a connection enters one of the accept queues, a start tag is associated with it. Its value depends on the weight assigned to the class this connection belongs to. The connections are then accepted in the increasing order of these tags. SFQ has a proven property that the rate of connections accepted from a class is proportional to its weight. The weight assignment is performed at user-level by the algorithm described below. To facilitate weight assignment, the kernel maintains for each class the measured queueing delay, request arrival rate, and request service rate.

In addition, the expected queueing delay of a class is determined by the request arrival rate λ of that class and its share of the processing resources. As discussed above, the share of resources given to each class is determined by its weight. For a particular request arrival rate, a class's queueing delay decreases when it is allocated more share of resources. Moreover, the effect of the resource share on the queueing delay becomes less prominent as a class receives more share of resources. When the request arrival rate changes, the relationship between the delay and the resource share also changes. To illustrate this, Figure 2 shows the share-delay curves corresponding to two different arrival rates. This nonlinear relationship is easily predicted from queueing theory.

The simplest feedback-based weight adaptation algorithm continuously keeps track of the operating point [1] of each class on such a curve, and approximates the small segment of the curve around the operating point with a line. In control theoretic terms, this slope is the process gain. It determines the change in output (class delay) as a function of the change in input (class weight). A weight adjustment is then calculated for each class based on this approximated linear relationship such that under the new weight, class output (delay) is set exactly equal to the set point. This algorithm is invoked every adaptation interval, which is a fixed predetermined quantity. Every time it is invoked, the algorithm executes the following steps (where k is the invocation number): [2]

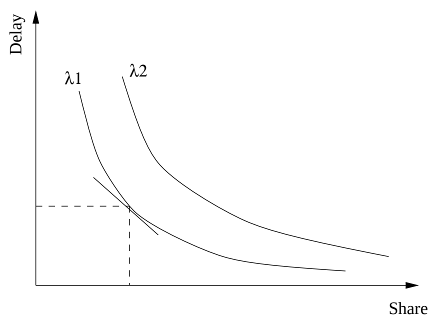


Figure 2: Relationship between Delay and Share

1. For each class i, query the kernel for its delay during the kth interval: $d_{i,k}$.
2. Since the request arrival rate is not constant, the measurement of the delay could be quite noisy. To avoid reacting to the noise and unnecessary weight adjustment, an exponential weighted moving average (EWMA) filter is introduced to smooth the measurement. For each class i, feed its delay $d_{i,k}$ into an EWMA filter. Denote the filter output as $D_{i,k}$. The expression for $D_{i,k}$ is:

$$D_{i,k} = \alpha \times D_{i,k-1} + (1-\alpha) \times d_{i,k} \tag{1}$$

where $D_{i,k-1}$ is the filter output for class i during the $k-1$ interval, and α is a configurable parameter controlling the smoothing effect.
3. For each class, calculate its slope as follows:

$$slope_i = \frac{D_{i,k} - D_{i,k-1}}{share_{i,k} - share_{i,k-1}}$$

where $share_{i,k}$ and $share_{i,k-1}$ are the shares for class i during the kth and $k-1$th interval, respectively.
4. For class i, calculate its share during the $k+1$th interval as follows:

$$share_{i,k+1} = share_{i,k} + \frac{goal_i - D_{i,k}}{slope_i} \tag{2}$$

[1]The operating point of the queue for one class depends on the arrival rate and the share of resource for that class.

[2]This is a simplified description. Please refer to [21] for more detail.

where $goal_i$ is the queueing delay goal for class i.
5. Notify the kernel to adjust the shares.

2.1 Analysis

In this section, we analyze the above adaptation algorithm from the perspective of control theory. In the analysis, we only focus on one class i, therefore omit the subscript i in the formulas. The algorithm can be modeled as a control loop as depicted in Figure 3. Recall that for a given class, the algorithm approximates its nonlinear share-delay relationship with piecewise linear segments. Therefore, we can model a queue around an operating point as a linear system. In the time domain, its model is:

$$delay = slope \times share + c$$

In the z-domain, the transfer function is $G_p = slope$. The weight adjustment (equation (2)) can be modeled by a classic integral controller, whose transfer function in the z-domain is:

$$G_c = K_I \times \frac{z}{z-1}$$

where

$$K_I = \frac{1}{slope}$$

The parameter estimator estimates K_I. And an EWMA filter (equation (1)) is equivalent to a 1st order low-pass filter, whose z-domain transfer function is:

$$G_f = z \times \frac{1-\alpha}{z-\alpha} \tag{3}$$

If we consider the system operating around an operating point, thus ignoring the parameter estimator, the closed-loop transfer function is:

$$\frac{G_c \times G_p \times G_f}{1 + G_c \times G_p \times G_f} = \frac{(1-\alpha)z^2}{(2-\alpha)z^2 - (1+\alpha)z + \alpha} \tag{4}$$

Notice that $G_p \times G_c = \frac{z}{z-1}$ and $slope$ is no longer in the closed-loop transfer function. Thus, if the slope estimation accurately describes a linearization around the operating point, then the closed-loop function only depends on α.

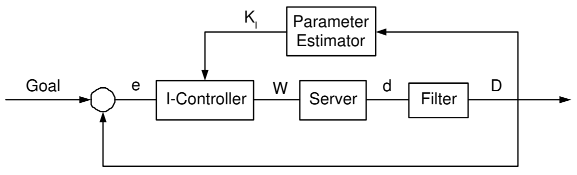


Figure 3: Block Diagram of the Original Algorithm

3 Transient Behavior

While the above control loop should eventually converge to the right resource allocation and achieve the desired delay, an experimental investigation reveals undesirable interactions between the feedback controller and the WFQ scheduler. Control of the WFQ scheduler is therefore given a closer look. In this section, we first study the effects of the two key parameters of feedback control of the WFQ scheduler; the α parameter of the EWMA filter and the adaptation interval. A departure is shown from theoretic expectations. The departure is explained by an interaction between the scheduler and the controller that leads to loop instability and general performance degradation. This interaction is modeled and analysed from a control theory perspective, followed by solutions that achieve good performance. We believe that the results, pitfalls, and insights illustrated in this section are directly applicable to any implementation of a WFQ element in which weights are dynamically adjusted using feedback control.

3.1 Experiment Setup

The experimental testbed consists of three machines connected by a 100Mbps Ethernet. The server is a 550MHz P-III machine with 256 MB RAM and runs the patched kernel. The server runs Apache web server 1.3.19, and the MaxClient parameter of Apache is set to 300 processes. Each client machine has a 550 MHz P-III CPU and 256 MB RAM, and runs Linux 2.4.7. SURGE [1] is used to generate the HTTP/1.1 workload. The requested content is dynamically generated by a CGI script at the server side. The CGI script first sleeps for some time before sending back the reply. The sleep time follows a uniform distribution U(0, 50ms). It simulates the time it takes to query the database or application server. The purpose is to stress the accept queue without loading the CPU.

We modify the SURGE so that the client number simulated can be dynamically changed. Throughout the experiment, two classes of clients are simulated. Originally, each class has 100 clients. During 100s to 200s, the client number of class 0 increases from 100 to 300, and remains at 300 for the rest of the experiment. The client number of class 1 does not change. The connection rates of the two classes are plotted in Figure 4. This workload simulates the abrupt traffic increase. The delay goals for the two classes are 1s and 20s, respectively.

3.2 Parameter of the EWMA filter

As we have seen in Section 2.1, the EWMA filter is equivalent to a 1st order low pass filter. Control theory tells us that 1st order low pass filters can introduce lags, thus slowing down the response of a system. The degree of the slowdown depends on the value of α. From Equation 3, we see that the filter adds a pole to the system at α, thus a smaller value of α will result in a faster system.

The problem we face is that α cannot be arbitrarily small, since a smaller α also lets more noise enter the system. The proper value should therefore be chosen based on the noise level, which is related to the variability in system load. System load in most Internet servers is highly variable, necessitating a fairly large α, and thereby resulting in a slower control system response.

To overcome this problem, we use a separate filter for the feedback and the parameter estimation. The filter for the feedback path has a smaller α: 0.3, to improve the responsiveness. The filter for the parameter estimation uses a larger α: 0.5, to reject noise. Figure 5 shows the new structure.

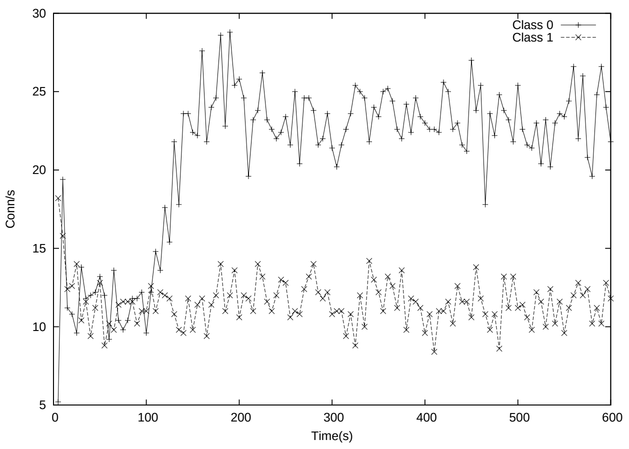


Figure 4: Connection Rate of Two Classes

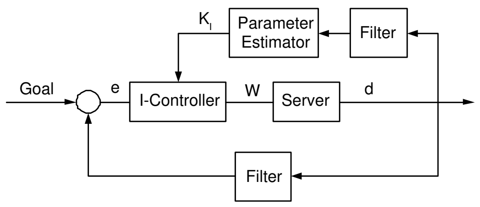


Figure 5: Separate Filter for the Feedback and Parameter Estimation

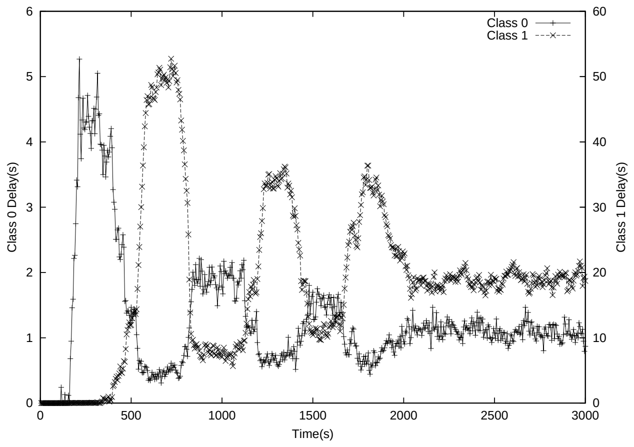
We then set the adaptation interval to 30s, and perform the experiment using the old and the new filter structure respectively. The behavior of the old and new system is plotted in Figure 6(a) and Figure 6(b) respectively. Comparing the two, the new system settles down much faster than the old one. The settling time is reduced from 2000s to 500s.

3.3 Adaptation Interval

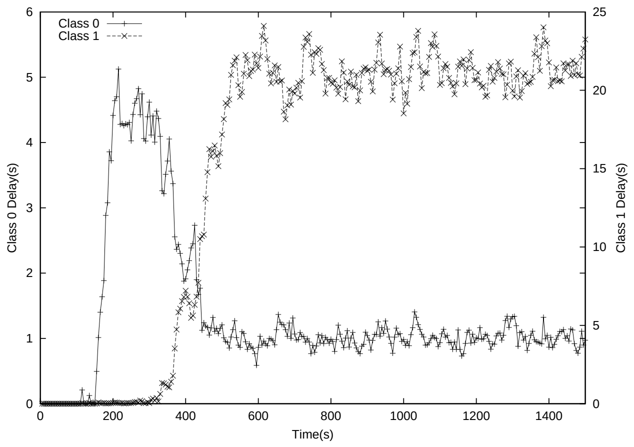
Another key parameters of the algorithm is the adaptation interval, which in control terminology is called the sampling period. As we have seen in the previous section, the settling time depends on the EWMA parameter α. In particular, α determines the number of sampling periods it takes for the system to settle. Expressed in absolute time units, the settling time is given by the number of periods S multiplied by the length of the period. Hence, given a fixed α, a smaller sampling period should, in principle, lead to a faster absolute settling time. We conduct an experiment to verify this claim.

Let us set the sampling period to a large value of 200s. As expected, this conservative setting results in a very slow transient response. As shown in Figure 7, it takes the system almost 1 hour (3500s) to settle down.

We now reduce the sampling period in the experiments. We are interested in two questions, namely, (1) Is the change of the settling time proportional to the change of the adaptation interval as postulated from theoretical analysis? For example, if the adaptation interval is reduced by half, will the settling time also be reduced by half? (2) What happens when the adaptation interval is very short? Does it introduce any



(a)Adaptation interval 30s with single filter



(b)Adaptation interval 30s with separate filters

Figure 6: Filter effect

side effects to the system? In other words, is there any lower bound on the adaptation interval?

We have already seen the result of using an adaptation interval of 30s in Figure 6(a). Not surprisingly, the smaller adaptation interval does improve the performance: the settling time is reduced to 2000s. But, this improvement is relatively small compared with the change of the adaptation interval. The adaptation interval is reduced to 1/6 of the original value (from 200s to 30s), but the settling time is only reduced by half.

Then, we repeat the experiment using a more aggressive adaptation interval: 5s. This time the system enters an oscillatory state and cannot settle down as shown in Figure 8. The experiment is repeated using the multi-filter structure. The system is still unstable.

Two interesting observations can be made from these experiment results: (1) Using a smaller adaptation interval does improve the system's responsiveness. But the improvement is limited. (2) There does exist some lower bound of the adaptation interval.

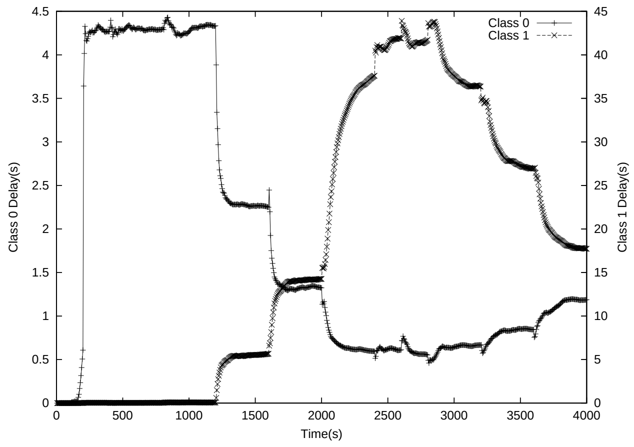


Figure 7: Adaptation Interval 200s with single filter

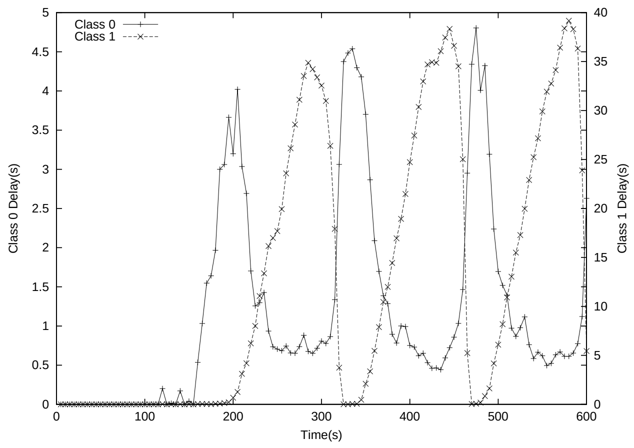


Figure 8: Adaptation Interval 5s with single filter

When the interval is too small, the system can become unstable. Both of these observations are at odds with Equation (4), which implies other hidden factors are affecting the system's stability.

3.4 Deadtime Effect

According to control theory, the stability of a system depends on the pole positions of its closed loop transfer function. Observe that in equation (4), only α appears in the denominator, which means that only α can affect the stability of the system. The fact that a small adaptation interval can lead to instability even if α is not changed implies that the system's model (i.e., the transfer function) changes in the case of a small adaptation interval. The closed loop system consists of four parts: an accept queue including its scheduler, an EWMA filter, an integral controller, and a parameter estimator. The behavior of the EWMA filer and the integral controller is well understood, therefore, we turn our attention to the accept queue, especially the scheduler.

The accept queue scheduler implements the SFQ algorithm. Recall that the SFQ algorithm schedules connections according to the values of the start tags. Since the tag is assigned when a connection first enters the queue and its value is decided by the weight of the class this connection belongs to at that moment, any further weight changes has no influence over the connections already in the queues. In other words, the new weight change will not take effect until the connections currently in the queues are all accepted. We call the time it takes the system to drain the backlog *deadtime*.

The existence of deadtime is exposed by a small experiment. The weights are initially 0.9 for class 0 and 0.1 for class 1. After the system stabilizes, at time 100s, the weights for both classes are changed to 0.5. The weight and delay for class 0 are plotted in Figure 9. Since the weight for class 0 drops, the delay experienced by class 0 increases. But the increase does not happen immediately after the weight is changed. Rather, it happens more than 10s after the weight is adjusted. This 10s delay is the deadtime.

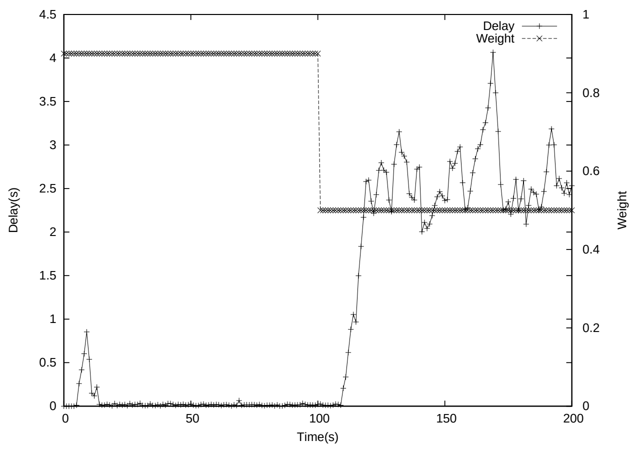


Figure 9: Evidence of Deadtime

Under normal situation, the deadtime is very short, thus does not become a problem. When the workload is high, the deadtime can be as long as several seconds as shown in the above small example. If the adaptation interval is comparable to the deadtime, the effect of the deadtime on the behavior of the system can no longer be ignored, thus must be taken into account.

Simulation is used to show the effect of the deadtime on the system. The same settings as those of the simulation in section 3.2 are used, and α is set to 0.3. The only difference is that now the server model is $-\frac{1}{16} \times \frac{1}{z^2}$. The extra term $\frac{1}{z^2}$ represents a deadtime of two adaptation intervals. The response of the system toward a step input is shown in Figure 10. The behavior clearly indicates an unstable state. According to control theory, a system is stable if its poles are all within the unit circle. Figure 11 compares the poles of the system with and without the extra $\frac{1}{z^2}$ term. When the extra term is added, the poles are moved from within the unit circle, which is the stable region, to the boundary of the unit circle, which is the unstable region.

How the deadtime gives birth to the oscillation can be intuitively explained as follows. Suppose that now the delay of class 0 is higher than its goal. The adaptation algorithm increases the weight for class 0. After 5 seconds, the adaptation algorithm

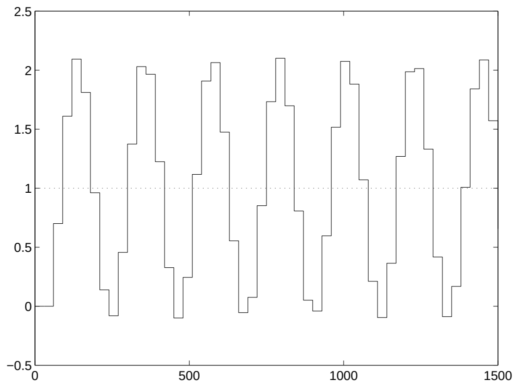


Figure 10: Simulate the Effect of Deadtime

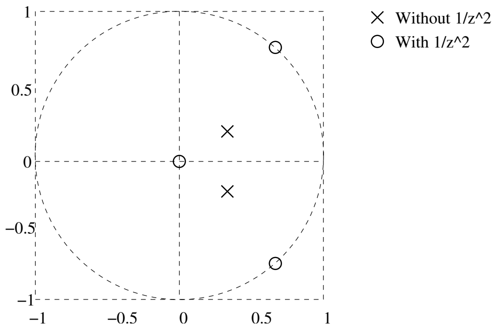


Figure 11: Pole Positions of Two Systems

expects to observe immediate result due to this weight increase, i.e., reduced queueing delay, and makes further adjustment based on the observation. Because of the deadtime, however, this expected delay reduction does not show up after 5 seconds. Therefore, the adaptation algorithm has to further increase the weight for class 0. This keeps happening until the queueing delay for class 0 drops. But by then, it is too late. All the weight increases applied have already had a cumulative effect on the system, and the weight for class 0 is much more than needed. The direct result is that class 1 begins to suffer. Then the same process happens to class 1 again. This essentially forces the system into an oscillatory state.

There are basically three ways to fix the problem. (1) Fix the implementation of SFQ such that the new weight change updates the start tags of those requests already in the queues. (2) Enrich the server model to take into account the deadtime. (3) Adopt variable adaptation intervals so that the deadtime becomes invisible to the adaptation algorithm. The first option is inefficient because it implies changing the logical timestamp of all enqueued requests at every sampling time, which introduces high overhead. The second option is viable, but requires more sophisticated control derived using more advanced results in control theory that pertain to systems with variable delay. We choose the third option due to its simplicity. It provides a straightforward solution that is able to maintain the simplicity of the feedback controller while achieving the needed performance.

To support variable adaptation intervals, one more per class measurement has to be maintained by the kernel: the accept queue length. When the adaptation algorithm changes the weight, it also records the current length of the queue for each class, i.e., the backlog size. After that, every second the adaptation algorithm queries the kernel for the number of requests accepted. It keeps querying until the backlog of all the queues are cleared, which means the end of the deadtime. Then it waits for the desired adaptation interval before it changes the weight again. Since the deadtime is not a fixed value, the actual adaptation interval is also variable: the deadtime plus the desired fixed adaptation interval, as shown in Figure 12.

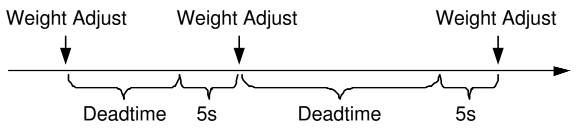


Figure 12: Variable Adaptation Interval

We set the desired adaptation interval to 5s, and repeat the experiment using this new algorithm. The response of the system is plotted in Figure 13. The most obvious improvement over Figure 8 is that the system is stable in this case. Compared with the case of 30s adaptation interval (Figure 6(b)), the settling time is reduced to 200s from 350s. This improvement is not as drastic as when the adaptation interval is changed to 30s from 200s. The reason is that the deadtime now becomes a major contributor to the settling time, and there is no way to reduce the deadtime unless we change the implementation of the scheduler. The latter option, however, may invalidate the set of convenience properties we know about the WFQ scheme, which represent the main advantages of that policy.

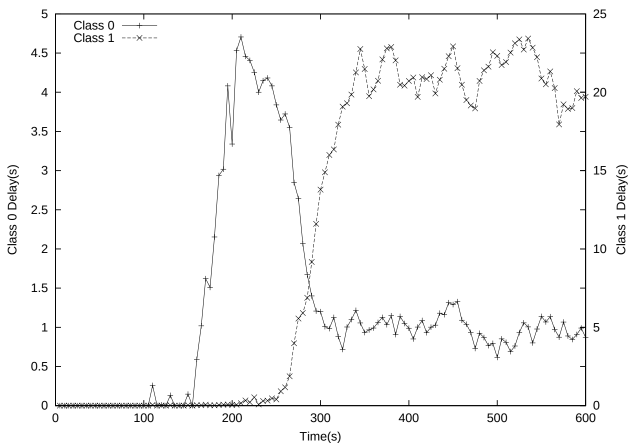


Figure 13: Variable Adaptation Interval with separate filters

4 Discussion and Future Work

The filter parameter α has dual effects on the system: smoothing the measurement and introducing delay into the system. Bigger value of α leads to a stronger smoothing effect and a longer delay. In the experiments, we pick the value offline based on the workload characteristics. In a more realistic scenario where the workload characteristics change quite often, the value has to be decided online using, for example, the fuzzy estimation approach outlined in [13].

SFQ is not the only scheduling algorithm that has the deadtime problem. WFQ [5] (also known as Packetized Generalized Processor Sharing (PGPS) [19]), FQS [9], Self Clocked Fair Queueing (SCFQ) [7], and WF^2Q [3] all work in a similar way as SFQ, therefore they all have the similar problem. We have proposed three solutions to overcome the deadtime. We choose the third one due to its simplicity. However, a more thorough solution is to change the way SFQ is implemented. Basically, when the weight assignment is changed, the tags associated with those connections already in the queues need to be recalculated. This re-calculation of the tags should preserve the property that the connection accept rate of one class is proportional to the weight of that class. One negative effect is that extra CPU cycles are needed for the re-calculation. We will implement this solution and evaluate the overhead it incurs.

In this paper, we only focus on two parameters' effects on the transient behavior of the original algorithm. The algorithm itself also worths some investigation. For example, the weight adjustment is essentially an integral controller. Integral controller is good because it can eliminate steady-state error, however, it can also result in worse transient behavior. We plan to implement other types of controllers, and compare their performance.

5 Related Work

Many researchers have successfully provided solutions for QoS guarantees using feedback control. For example, a differentiated web cache service is presented in [18], [17]. The service is capable of providing different hit ratios for different classes of contents. In [20], control theory is used to control the length of the RPC queue in a Lotus Notes server. Control theory is applied in [15] to provide delay guarantee in web servers. The number of processes servicing each class is dynamically adjusted to achieve desired delay goal.

An architecture is proposed in [4] that maintains separate service queues for different classes of clients. While it is capable of providing preferential services to premium clients, it cannot provide any guarantee on the service. A scheme named Latency-targeted Multiclass Admission Control(LMAC) is developed in [12]. No interaction among low level system resources are modeled. Instead, system resources are abstracted into high-level virtual server. The algorithm uses measurements of requests and service latencies to control each class's quality of service.

In web based business applications, workload of different nature can be sent to a web server. The change in workload nature and volume can cause the bottleneck resource to shift. An adaptive QoS framework is introduced in [21] to provide system-wide QoS guarantee even when the bottleneck resource shifts. This is achieved by managing multiple resources simultaneously. Our work only focus on one part of the framework: the accept queue management.

A similar methodology is applied in [10] to the analysis and design of Active Queue Management control systems using random early detection (RED) scheme. The authors first linearize a previously developed nonlinear dynamic model of TCP. Based on this linearized model, the stability condition of the system is discussed in terms of network parameters such as link capacity and load. The authors then design two alternative AQM controllers, both of which respond faster that the RED controller [11].

In the context of GPS queues with a dual leaky-bucket rate control, there exists previous work in determining the per-class weights. [6] analyzed the two-class case, and it was generalized to the multi-class case in [14]. These approaches aim to achieve QoS guarantees in a statistical sense. However, they assume knowledge of the service capacity (which in general may not be known) as well as the leaky-bucket traffic assumption and a more detailed QoS specification involving not only delay bound but also loss probabilities. Moreover, the dynamics of the actual scheduling mechanism are not taken into account, which may affect the applicability of the methods.

6 Conclusion

In this paper, we examine the transient behavior of an adaptive system for QoS guarantees for web servers. This system is essentially a WFQ scheduler where the weights can change dynamically. We show that it is important to properly configure its two important parameters so that the system exhibits good responsiveness; we have improved by an order of magnitude over the previously published result for such a mechanism. We also uncover how a flawed implementation of the SFQ algorithm can damage the system stability, and propose a solution. Basic insights from control theory are used to explain the results of the experiments. This paper is intended as a case study. We believe that other feedback based adaptive algorithms could also be analyzed and improved in a similar fashion. Further, the results here could be applied to tune other applications where an adaptive WFQ type algorithm is used.

References

[1] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[2] L. Benmohamed and S. M. Meerkov. Feedback control of congestion in packet switching networks: the case of a single congested node. *IEEE Transactions on Networking*, 1(6):693–708, Dec. 1993.

[3] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM*, 1996.

[4] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, pages 64–71, September 1999.

[5] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair-queueing algorithm. In *ACM SIGCOMM*, 1989.

[6] A. Elwalid and D. Mitra. Design of generalized processor sharing schedulers which statistically multiplex heterogeneous QoS classes. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'99)*, pages 1220–1230, New York, NY, USA, Mar. 1999.

[7] S. Golestani. A self-clocked fair queueing scheme for high speed applications. In *Proceedings of IEEE INFOCOM*, pages 636 – 646, 1994.

[8] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated servicespacket switching networks. In *ACM SIGCOMM*, 1996.

[9] A. G. Greenberg and N. Madras. How fair is fair queuing? *Journal of the ACM*, 39(3):568 – 598, July 1992.

[10] C. V. Hollot, V. Misra, D. F. Towsley, and W. Gong. A control theoretic analysis of RED. In *Proceedings of IEEE INFOCOM*, pages 1510–1519, 2001.

[11] C. V. Hollot, V. Misra, D. F. Towsley, and W. Gong. On designing improved controllers for AQM routers supporting TCP flows. In *Proceedings of IEEE INFOCOM*, pages 1726–1734, 2001.

[12] V. Kanodia and E. Knightly. Multi-class latency-bounded web services. In *the 8th International Workshop on Quality of Service*, 2000.

[13] S. Keshav. A control-theoretic approach to flow control. In *ACM SIGCOMM*, 1991.

[14] K. Kumaran, G. Margrave, D. Mitra, and K. R. Stanley. Novel techniques for design and control of generalized processor sharing schedulers for multiple QoS classes. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, pages 932–941, Tel Aviv, Israel, Mar. 26–30 2000.

[15] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A feedback control architecture and design methodology for service delay guarantees in web servers. In *IEEE Real-Time Technology and Applications Symposium*, June 2001.

[16] Y. Lu, T. Abdelzaher, C. Lu, L. Sha, and X. Liu. Feedback control with queueing-theoretic prediction for relative delay. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.

[17] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for qos guarantees and its application to differentiated caching services. In *the 10th International Workshop on Quality of Service*, May 2002.

[18] Y. Lu, A. Saxena, and T. F. Abdelzaher. Differentiated caching services: A control-theoretical approach. In *International Conference on Distributed Computing Systems*, April 2001.

[19] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking (TON)*, 1(3), June 1993.

[20] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *Real Time System Journal*, 23(1-2), 2002.

[21] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and P. Shenoy. An observation-based approach towards self-managing web servers. In *the Tenth International Workshop on Quality of Service*, 2002.

[22] L. Sha, X. Liu, Y. Lu, and T. Abdelzaher. Queuing model based network server performance control. In *IEEE Real-Time Systems Symposium*, December 2002.