# Request Scheduling for Differentiated QoS at Website Gateway

*Ying-Dar Lin[1], Ching-Ming Tien[1], Shih-Chiang Tsao[1], Shuo-Yen Wen[1], Yuan-Cheng Lai[2]*
*[1]Department of Computer Science, National Chiao Tung University*
*[2]Department of Information Management, National Taiwan University of Science and Technology*
*Taiwan, R.O.C.*
*{ydlin, cmtien, weafon, sywen@cs.nctu.edu.tw}, laiyc@cs.ntust.edu.tw*

## Abstract

With the explosive growth of Web traffic, the load on a Web server becomes heavier, leading to the longer user-perceived latency. Website operators would like to employ service differentiation to offer better throughput and shorter user-perceived latency to some specific users. This paper presents an HTTP request scheduling algorithm deployed at the Website gateway to enable the Web quality of service without any modification to client or server software. A variation of the deficit round robin algorithm for packet scheduling and a window control mechanism are presented to decide the order and the releasing time of requests, respectively. The order is decided by the response size of the requests and the pre-defined service weights. The ratio of the service rate got by the service classes is determined by the weights, whereas the releasing time is decided by the service rate of the Web server. The evaluation shows the scheduling algorithm can provide service differentiation and improve server throughput and user-perceived latency. When the weight ratio 6:3:1 is assigned to three service classes, the QoS Website gateway makes them get 60%, 30%, and 10% of the overall throughput as expected, regardless whatever page sizes. In addition, the throughput and the user-perceived latency of the class with the largest weight can be improved by up to 176% and 69% of the QoS-disabled values, respectively.

**Keywords:** Web QoS, service differentiation, request scheduling, gateway.

## 1 Introduction

Today more and more users connect to the Internet to surf the World Wide Web. The more accesses to a website, the heavier load will be on the Web server. The busier server leads to the longer user-perceived latency, which means a user will wait for a longer time to download a Web page. Therefore, website operators would like to improve user-perceived latency to keep their customers.

To reduce the user-perceived latency, the bottleneck of accessing a Web page should be identified first. An example of user-perceived latency measurement on downloading the homepages of 40 important US-based business Websites from Keynote in 2001 is given here. Keynote Company measured website performance from its 1,700 measurement computers distributed in 50 metropolitan areas worldwide [1]. The user-perceived latency, i.e. Web page download time, is decomposed into six components: 1) Domain Name System (DNS) lookup time, 2) Transmission Control Protocol (TCP) connection time, 3) server delay time, 4) redirection time, 5) base page download time, and 6) content download time. The report of Keynote said the longest duration of the download time is in transferring the content, and this part is mainly determined by the congestion statuses of the network and the server.

The network bottleneck could be resolved by employing network Quality of Service (QoS) mechanisms [2][3], whereas the server bottleneck could be resolved by clustering servers, caching Web pages, and so on. However, network QoS is hard to be deployed in nowadays Internet infrastructure because all routers have to support and enable network QoS protocols, e.g. Resource reSerVation Protocol (RSVP) [4]. At the server side, the HyperText Transfer Protocol (HTTP) traffic can be controlled at the packet level or the application level. Several recent research have proposed application-level QoS [5-10] to provide service differentiation because this approach provides more flexible policies to website operators in traffic control. They made efforts on modifying the system kernel [5] or the daemon program [5-10] of a Web server to provide Web QoS. However, the shortcoming is those mechanisms are operating system or server daemon dependent.

This research focuses on resolving the server bottleneck because website operators can completely control their servers, but cannot do much on improving the whole network performance. The goals are to improve the server throughput and reduce the user-perceived latency for high-class users, that is, to provide service differentiation at the server side, and thus allow these users to perceive the shorter latency on downloading Web pages.

A QoS website gateway, independent of operating systems and server daemons and transparent to clients and servers, is presented in this paper. HTTP requests incoming to the gateway will be classified and queued into different class queues by the application-level inspection. A variation of Deficit Round Robin (DRR) scheduling [11] and a window control mechanism are presented to decide the order and the releasing time of requests, respectively. In addition, a server probing mechanism is used to seize the characteristics of Web pages, such as Universal Resource Locator (URL) and the response size, to help the request scheduling.

The rest of this paper is organized as follows. Section 2 describes the architecture of a QoS website gateway and presents the design of the request classifier, the request scheduling algorithm, and the probing mechanism. The implementation and evaluation of the QoS website gateway are discussed in Section 3. Finally, the conclusion and the future work are given in Section 4.

## 2 QoS Website Gateway Architecture and Request Scheduling Algorithm

Given a Web server and several classes of clients, the goal is to provide service differentiation by HTTP request scheduling on the website gateway. For concentrating on the design of the request scheduling, the server cluster is not considered in this research. In server cluster scenarios, the issues of the server load balancing also need to be considered. Therefore, a single Web server is the final scenario discussed in this research.

### 2.1 QoS Website Gateway Architecture

The architecture of the QoS website gateway is shown in Figure 1. All HTTP requests originated from clients pass through the gateway, and the gateway schedules them to the Web server according to the QoS policies and the service rate of the Web server. The request classifier first classifies the incoming requests into different service classes by inspecting the content of IP headers, HTTP headers, and payloads. The classified requests are then queued into the class queues. The request scheduler decides which request should be fetched next from a class

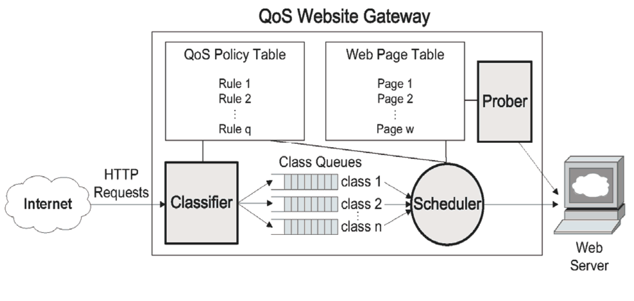


Figure 1 Architecture of the QoS Website

queue and when the request should be released to the Web server according to the QoS policy table and the service rate of the Web server, respectively. For knowing the characteristics of Web pages stored in the Web server, the server prober probes the Web server before the on-line operation of the gateway. In a word, request classification, request scheduling and server probing are the three things the website gateway does for the service differentiation. The more details are discussed as follows.

### 2.2 Request Classification

A common classification paradigm is to inspect the IP 5-tuples (source IP address, destination IP address, source port number, destination port number, and protocol type) of a packet header. However, this type of classification is content-blind; that is, the classifier cannot see the information contained in the application layer protocols. The website operator may wish to define more flexible QoS policies based on the application layer protocols such as HTTP for the service differentiation. Therefore, the classifier should be content-aware, that is, it sees the information contained in the protocol headers and payloads.

The purpose of the request classifier is to classify the incoming requests into proper classes based on the QoS policy table. The rules in the policy table can be defined according to the information contained in IP packet headers, HTTP headers and HTTP payloads. HTTP headers generally contain URL, User-Agent, Content-Length, etc., whereas HTTP payloads generally contain cookie names, Secure Socket Layer Identification (SSL) IDs, etc. The request classifier compares the information contained in the incoming requests with the rules in the QoS policy table. If a request matches a specific rule of a service class, it will be put into the corresponding queue and wait for being scheduled.

### 2.3 Request Scheduling

In this research, we present a DRR-based request scheduling. The goal of the request scheduler is to allocate the output throughput of responses on the Web server for different classes of users by scheduling their requests. After requests have been classified and queued in the corresponding queues, the request scheduler should decide which request should be fetched next and when the request should be released to the Web server.

DRR is an packet scheduling algorithm with O(1) complexity for the selection of the next packet. In DRR, for the differentiation, each class queue is given a deficit counter (DC) and assigned a specific quantum in bytes. DRR selects packets from each queue in a round-robin manner. Every time when a queue is in served, its DC would be added by its quantum. Next, the head-of-line (HOL) packet in the queue will be sent out if the DC is larger than the length of the packet, and then the DC will be decreased by the length of the packet. Such a packet transmission would be repeated until the value in DC is smaller than the length of the HOL packet. Finally, the residual value of the DC would be accumulated for the class to use in the next round.

Although DRR is an ideal packet scheduling, two modifications on DRR are necessary for scheduling the requests and allocating the throughput of responses on the Web server. First, the value of DC should be decreased by or compared with the length of the responses but not the requests, because the target to be allocated is the throughput of output responses from the Web server. Second, unlike the packets consequently sent under DRR, the requests cannot be sent out closely following the last request. The releasing rate of requests should be throttled such that the released request would not overwhelm the processing capacity of the Web server.

For this, a window control mechanism is presented to adjust the releasing rate, as shown in Figure 2. The window size stands for the number of the maximum concurrent transmissions of response between the gateway and the Web server. When a request is released, the window value is decremented by one. Conversely, when a resulting response has passed through the gateway, the window value is incremented by one. The scheduler checks the window before releasing a request. If the window value is large than zero, the scheduler releases the scheduled requests to the Web server; otherwise, it stops releasing the requests and waits until the window value is not zero. In this way, the processing capacity of the Web server can be utilized well without being overwhelmed. A small number (less than ten) is suggested to be assigned to the window size because a large window size may lead to an over-loaded server.

## 2.4 Server Probing

The request scheduler needs the response sizes of Web pages stored in the Web server when performing the scheduling.

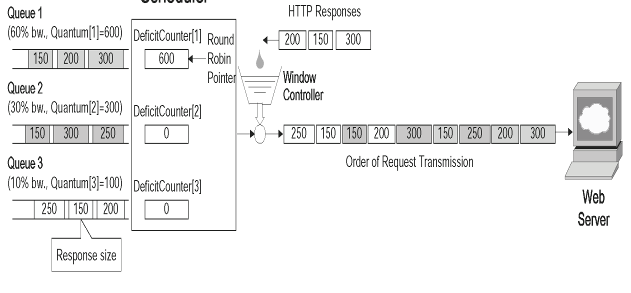


Figure 2 Request Scheduler and Window Controller

To get the response size of the Web pages, a popular method is running a program in the Web server to collect the sizes of the files storing the Web pages. However, the method has three drawbacks. First, developing multiple versions of programs is necessary for Web servers with different software packages and OSs. Second, Web hosts may not prefer running additional programs in their Web servers which may risk the stability of their servers. Third, the response sizes of the dynamic pages are irrelevant to the sizes of these pages and can be known only after these pages are requested and their responses are generated.

Therefore, this work employs a server prober at the gateway to probe the URL and the response size of each Web page on the server before the on-line operation of the gateway. The probed results are recorded in the Web page table and fed to the request scheduler. For probing the URL and the response size of each Web page on the server, the server prober first retrieves the homepage of the website, parses the homepage to find the embedded objects and the other hyperlinks. The prober recursively scans the Web pages within the same server link by link until all Web pages have been scanned. The probed URL and the response size of each page will be recorded in the Web page table, and they are mainly used for the initial accesses of the Web pages. Because the Web pages and the embedded objects on the server are assumed to be static, each URL and the corresponding response size is one-to-one mapping. The Web page table will be repeatedly updated by the later accesses of the Web pages. By this way, if the content of a Web page is changed in the future, i.e. the page size is changed; the request scheduler can update the Web page table because it knows the latest response size when receiving this page from the Web server.

## 2.5 Request Scheduling Algorithm

The pseudo code of the scheduling algorithm is shown in Table 1. Initially, all deficit counters are set to zero. Upon arrival of a request, the *Enqueuing Module* invokes *Classify*() and *Enqueue*() to classify the request and enqueue it to the corresponding queue, respectively. The *ActiveList* is used to avoid the overhead of examining empty queues. It maintains a list of indices of the active queues containing at least one request. In the *Dequeuing Module*, the active class queues are processed from the head of the *ActiveList*, say the class *i*. The scheduling algorithm fetches requests from queue $Q_i$ when there is enough service quantum and the window $w$ is not zero. The service quantum $DC_i$+$Quantum_i$ determines how many requests can be fetched from the $Q_i$, that is, the sum of the response sizes of the fetched requests cannot be greater than this service quantum. Before fetching and releasing a

Table 1 Request Scheduling Algorithm.

```
Incoming Request Part
Initialization:
For (i = 0; i < NumofClasses; i = i + 1)
  DC_i = 0;
Enqueuing Module: on arrival of request req
  i = Classify(req);
  If (ExistsInActiveList(i) == FALSE) then
    /* add queue i to active list */
    InsertTailActiveList(i);
    DC_i = 0;
  Enqueue(i, req); /* enqueue request req to queue i */
Dequeuing Module: once when ActiveList turns in nonempty
do
  i = RemoveHeadActiveList();
  DC_i = DC_i + Quantum_i;
  While ((DC_i > 0) and (Q_i is not empty)) do
    /* get request req from queue i */
    req = Head(Q_i);
    ResponseSize = GetSize(req);
    If (ResponseSize <= DC_i) then
      If (w != 0) then
        Send(Dequeue(Q_i));
        DC_i = DC_i − ResponseSize;
        w = w − 1;
      Else /* return to the original condition */
        InsertHeadActiveList(i);
        DC_i = DC_i − Quantumi;
        return(); /* exit this module */
    Else
      break; /* skip the while loop */
  If (Q_i is empty) then
    DC_i = 0;
  Else
    InsertTailActiveList(i);
  While (ActiveList is not empty)
Outgoing Response Part
Enqueuing Module: on arrival of response rsp
  Enqueue(rsp);
Dequeuing Module:
  While (TRUE) do
    If (Q is not empty) then
      Send(Dequeue(Q));
      w = w + 1;
```

request, i.e. invoking *Send(Dequeue($Q_i$))*, the scheduling algorithm checks if the *w* is not zero. If the *w* is not zero, the scheduling algorithm releases the request and decrements the *w* by one. Otherwise, the scheduling algorithm will not release any requests in the $Q_i$. After a resulting response has passed through the gateway, the *w* will be incremented by one.

## 3 Implementation and Evaluation

### 3.1 QoS Website Gateway Implementation

The request classifier and the request scheduler are implemented as a daemon program called "WebQ" on the NetBSD [12] system. Due to the small memory size in most gateway devices, the WebQ does not fork any child process when accepting a request for scalability. The single process invokes the *select*() system call to handle all socket descriptors concurrently. The WebQ runs at the user space and listens on the port 880 of the loopback IP address, i.e. 127.0.0.1:880, as shown in Figure 3. To make the WebQ work transparently to both clients and the Web server, the ipnat [13] utility rewrites the destination IP address and the port number of the incoming HTTP packets to redirect the requests to the WebQ for service differentiation. The WebQ performs the request classification and the request scheduling and sends the requests to the Web server. The HTTP responses from the Web server also pass through the WebQ and return to the clients. The prober is implemented as another daemon program which probes the characteristics of Web pages from the Web server before the on-line operation.

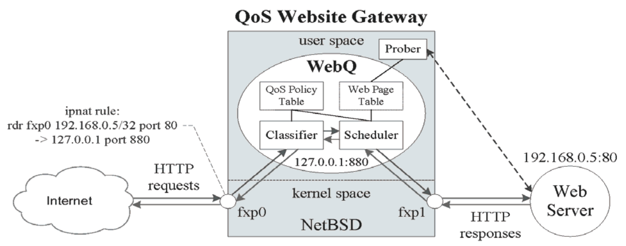


Figure 3 Implementation of the QoS Website

## 4 Performance Evaluations

The effect of the service differentiation can be evaluated on both the throughput and user-perceived latency. The aggregated throughput and the user-perceived latency of each service class are measured for comparing the effects between the activation and the deactivation of the request scheduling. The measurement is performed with fixed-sized and mixed-sized Web pages to demonstrate the robustness of the request scheduling algorithm. In addition, the effect of the window size is also evaluated.

### 4.1 Evaluation Environment

The evaluation environment consists of an Apache

Web server [14], the WebQ gateway, and several computers running the WebBench Web performance testing tool [15], as shown in Figure 4. The WebBench controller emulates multiple WebBench clients in the WebBench Hosts and then orders these clients to issue HTTP requests to the Web server and gathers the resulting data from the WebBench clients. The WebBench client issues a new request after it has completely received a response from the server. This means the sending rate of clients depends on the processing rate of the server. In this evaluation, the WebBench clients are divided into three service classes, whose ratio of the quanta is set to 6:3:1.

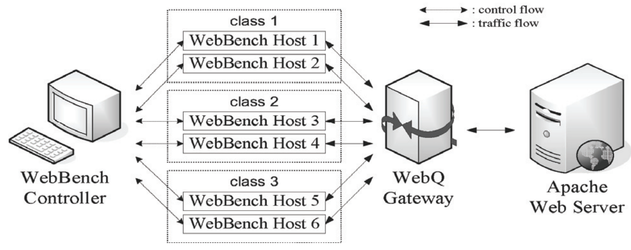
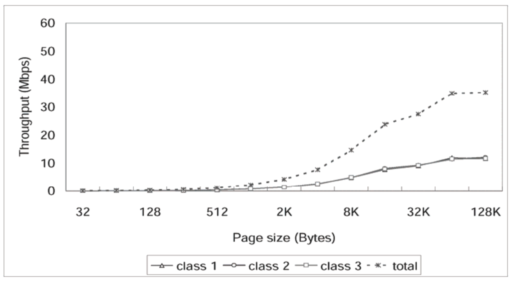


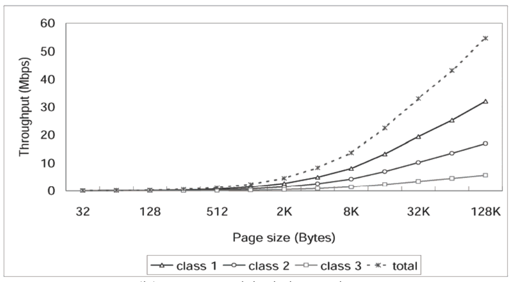
Figure 4 Evaluation Environment

## 4.2 Evaluation with Fixed-Size Web Pages

The evaluation with fixed-size Web pages is to observe the effects of the page size, which is changed from 32 bytes to 128K bytes. The resulting throughputs are shown in Figure 5(a) and 5(b), in which the throughput increases with the page size. The increase of the page size leads to the higher aggregated response size of the requested pages, i.e. throughput. In Figure 5(a), under the QoS-disabled case, the three service classes get the almost same throughputs because their requests have the same probability of entering the server. Nevertheless, in Figure 5(b), under the QoS-enabled case, the three service classes get the expected throughputs. The larger weight a service class has, the higher throughput this class gets. In addition, the throughput of the class with the largest weight is improved by up to 176% when the page size is 128K bytes, while that of the class with the smallest weight is penalized by 52%. Furthermore, the average of the total throughput 14.2 Mbps under the QoS-enabled case is higher than 11.7 Mbps under the QoS-disabled case because the request scheduling throttles the releasing request rate to avoid overwhelming the server.



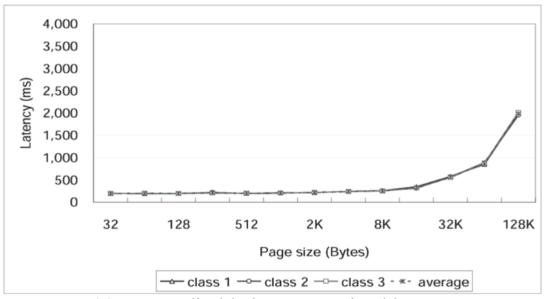
(a) QoS-disabled throughput



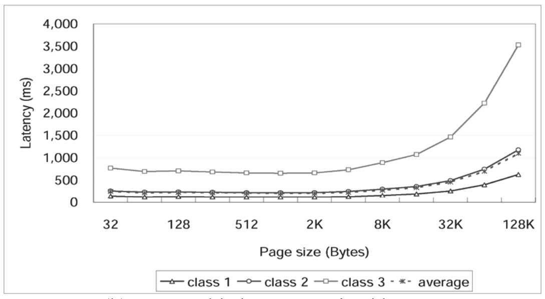
(b) QoS-enabled throughput

Figure 5 Throughput under Various Fixed-size Web Pages

The user-perceived latencies under the two cases are also compared, as shown in Figure 6(a) and 6(b). The user-perceived latency increases with the page size because the gateway has to process more packets for each response. The three service classes get the same user-perceived latency when the QoS is disabled, whereas they perceive different latencies when the QoS is enabled. The larger weight a service class has, the shorter latency this class obtains under the QoS-enabled case. In addition, the user-perceived latency of the class with the largest weight is improved by up to 69% when the page size is 128K bytes, while that of the class with the smallest weight is penalized by 75%. Note that the average of the user-perceived latency 351 ms under the QoS-enabled case is shorter than 440 ms under the QoS-disabled case. This proves the presented scheduling algorithm eliminates the server bottleneck and improves the total throughput.



(a) QoS-disabled user-perceived latency



(b) QoS-enabled user-perceived latency

Figure 6 User-perceived Latency under Various Fixed-size Web Pages

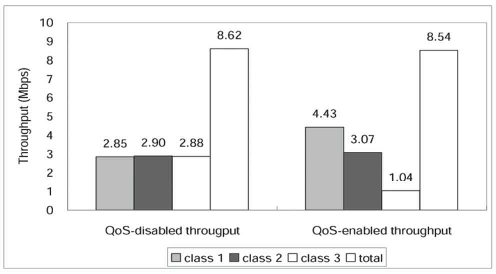
### 4.3 Evaluation with Mixed-Size Web Pages

In order to evaluate the QoS website gateway in a more realistic environment, the mixed-size Web pages are employed on the server. The page sizes have a lognormal distribution [16], whose probability density function is shown as follows:

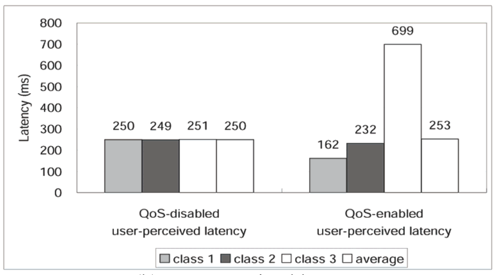
$$P(x) = \frac{1}{S\sqrt{2\pi x}} e^{-(\ln x - M)^2/(2S^2)},$$

where $S$ (standard deviation for the natural logarithm of the data) and $M$ (mean for the natural logarithm of the data) are set to 9.357 and 1.318, respectively.

The throughput of each class is shown in Figure 7(a). The ratio of the throughputs under the QoS-enabled case is still as expected, close to 6:3:1, demonstrating the request scheduling algorithm works well even in a more realistic environment. The user-perceived latency is shown in Figure 7(b). The observations on the evaluation with mixed-size Web pages are similar to that with fixed-size Web pages.



(a) Throughput



(b) User-perceived latency

Figure 7 Throughput and User-perceived Latency under Mixed-size Web Pages

### 4.4 Evaluation with Various Window Sizes

To see the effect of the window size on the service differentiation, the window size is changed from 1 to 100 during the measurement. The size of Web pages is set to 2K bytes. The total throughput under various window sizes is shown in Figure 8, in which the total throughput maintains the same level. This level (about 4 Mbps) is the maximum throughput of the server under the 2K-byte Web pages and would not be changed, regardless whatever window sizes. It reveals that the server is not overwhelmed when processing up to 100 concurrent connections in the evaluation environment, and furthermore, the aggregated request-sending rate is almost the same under these window sizes.

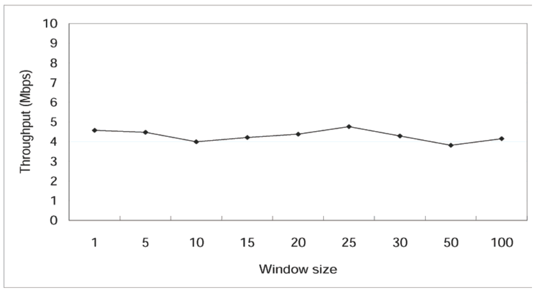


Figure 8 Total Throughput under Various Window Sizes

The effect of the window size on the throughput ratio is shown in Figure 9, in which the throughput ratio changes from 6:3:1 (60%:30%:10%) towards 1:1:1 (33.3%: 33.3%: 33.3%). The effectiveness of the service differentiation decreases with the increase of the window size. This is because the larger the window size, the more requests will be queued at the server instead of the gateway. The service differentiation cannot be carried into full effect without enough requests queued at the gateway. The above conclusion can be also applied to the effect of the window size on the latency, as shown in Figure 10.

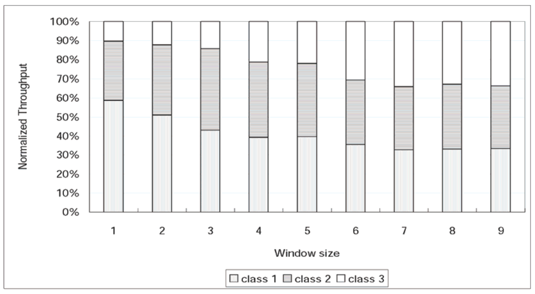


Figure 9 Throughput Ratio under Various Window Sizes

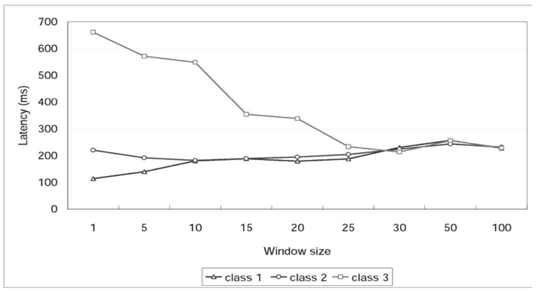


Figure 10 User-perceived Latency under Various Window Sizes

In summary, the setting of the window size does not matter when the server is not overwhelmed by the requests. However, the larger window size may cause the less effect of service differentiation under a small request-sending rate. Therefore, a small number is suggested to be assigned to the window size.

## 5 Conclusions

Service differentiation is a way for website operators to provide better throughput and shorter user-perceived latency to some specific users. This paper presents a request scheduling algorithm deployed at a website gateway to enable the Web QoS without any modification to client or server software. The QoS website gateway consists of a request classifier, a request scheduler, and a server prober. The content-aware request classifier classifies and queues incoming HTTP requests into corresponding class queues according to the pre-defined QoS policies. The request scheduler and the window control mechanism decide which request should be fetched next and when the request should be released to the Web server. The server prober scans URLs, gets the corresponding response size of the Web pages on the server, and feeds the probed results to the request scheduler for helping the scheduling.

The QoS website gateway is evaluated in the scenarios of fixed-size Web pages and mixed-size Web pages to demonstrate the robustness of the request scheduling algorithm. The results show the throughput and the user-perceived latency of the class with the largest weight can be improved by up to 176% and 69% of the QoS-disabled values, respectively. The total server throughput is also improved.

The future work is to enable service differentiation at the QoS website gateway for a server cluster. In this case, the QoS website gateway has to schedule the requests for the service differentiation and balance the server load simultaneously.

## Acknowledgment

This work is supported in part by National Science Council Program, Cisco, Intel, and Chung-Hua Telecom.

## References

[1] P. Mills and C. Loosley, *A performance Analysis of 40 e-Business Web Sites, CMG Jour. Computer Resource Management*, Issue 102, Spring 2001.

[2] R. Braden, D. Clark, and S. Shenker, *Integrated Services in the Internet Architecture: an Overview, IETF RFC 1633,* Jun. 1994.

[3] S. Blake, D. Black, et al., *An Architecture for Differentiated Services, IETF RFC 2475,* Dec. 1998.

[4] R. Braden, Ed., L. Zhang, et al., *Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification,* IETF RFC 2205, Sep. 1997.

[5] J. Almeida, M. Dabu, A. Manikutty, and P. Cao, *Providing Differentiated Levels of Service in Web Content Hosting, Proc. 1998 Workshop on Internet Server Performance*, Jun. 1998.

[6] R. Pandey, J. F. Barnes, and R. Olsson, *Supporting Quality of Service in HTTP Servers, Proc. 17th Annual ACM Symp. Principles of Distributed Computing*, Jun. 1998, pp. 247-256.

[7] L. Eggert and J. Heidemann, *Application-Level Differentiated Services for Web Servers, World Wide Web Jour.*, vol. 2, no. 3, Aug. 1999, pp. 133-142.

[8] N. Bhatti and R. Friedrich, Web Server Support for Tiered Services, *IEEE Network Mag.*, vol. 13, issue 5, Sep.-Oct. 1999, pp. 64-71.

[9] N. Vasiliou and H. Lutfiyya, *Providing a Differentiated Quality of Service in a World Wide Web Server, ACM SIGMETRICS Performance Evaluation Review*, vol. 28, issue 2, Sep. 2000, pp. 22-28.

[10] X. Chen and P. Mohapatra, *Performance Evaluation of Service Differentiating Internet Servers, IEEE Trans. Computers*, vol. 51, issue 11, Nov. 2002, pp. 1368-1375.

[11] M. Shreedhar and G. Varghese, *Efficient Fair Queuing Using Deficit Round-Robin, IEEE/ACM Trans. Networking*, vol. 4, issue 3, Jun. 1996, pp. 375-385.

[12] The NetBSD Project, http://www.netbsd.org/.

[13] IP Filter, http://coombs.anu.edu.au/~avalon/.

[14] The Apache HTTP Server Project, http://httpd.apache.org/.

[15] WebBench,http://www.veritest.com/benchmarks/web bench/.

[16] P. Barford and M. Crovella, *Generating Representative Web Workloads for Network and Server Performance Evaluation, ACM SIGMETRICS Performance Evaluation Review*, vol. 26, issue 1, Jun. 1998, pp. 151-160.

## Biographies

**Ying-Dar Lin** is a professor of Computer Science at National Chiao Tung University, where he is also the director of Computer and Network Center and Network Benchmarking Lab (NBL). His research interests include quality of services, deep packet inspection, and hardware software co-design. He is spending his sabbatical year, from July 2007, at Cisco, San Jose. Dr. Lin graduated from National Taiwan University (B.S. 1988) and UCLA (Ph.D. 1993).

**Ching-Ming Tien** was born in Hsinchu, Taiwan, in 1975. He received the B.S. and M.S. degrees in Industrial Education from National Taiwan Normal University, in 1998 and 2000, respectively. His research interests include Web quality of service, content delivery networks, bridging and routing protocols. He can be reached at cmtien@cis.nctu.edu.tw.

**Shih-Chiang Tsao** was born in Hsinchu, Taiwan in 1975. He received the B.S. and M.S. in Computer Science from National Chiao Tung University in 1997 and 1999, respectively. From 1999 to 2003, he worked as an associate researcher in Chung-Hwa Telecom. He received the Ph.D. degree in Computer Science from National Chiao Tung University in 2007 and is doing a post-doc in Lawrence Berkeley National Laboratory since May 2008. His research interests include TCP-friendly congestion control algorithms, fair-queuing algorithms, and Web QoS. He can be reached at weafon@cs.nctu.edu.tw.

**Shuo-Yen Wen** was born in Taipei, Taiwan in 1980. He received the B.S degree in Computer Science from National Chengchi University in 2002, and the M.S degree in Computer Science from National Chiao Tung University in 2004. His research interests include Web quality of service, and technology and innovation management. He can be reached at sywen@cis.nctu.edu.tw.

**Yuan-Cheng Lai** received the Ph.D. degree in Computer Science from National Chiao Tung University in 1997. He joined the faculty of the Department of Information Management at National Taiwan University of Science and Technology in 2001 and has been an associate professor since 2003. His research interests include wireless networks, network performance evaluation, network security, and content networking.