國立交通大學

資訊科學系

碩士論文

在網站閘道器上提供差別服務品質之請求排程

Request Scheduling for Differentiated QoS at Website Gateway

研 究 生：溫碩彥

指導教授：林盈達 教授

中華民國 九十三 年六月

在網站閘道器上提供差別服務品質之請求排程

學生：溫碩彥　　　　指導教授：林盈達

國立交通大學資訊科學系

摘 要

隨著網站流量的大幅成長，網站伺服器的負載也變得愈重，進而造成使用者感受到的等待時間也愈長，網站經營者希望能夠採用差別服務來改善特定使用者得到的網站吞吐量以及縮短其感受到的等待時間。本文提出一個部署在網站閘道器上的 HTTP 請求排程演算法來提供差別服務品質，其不需要修改任何在客戶端或伺服器端的軟體。以閘道器輸出連結的觀點來看，處理請求的時間是根據其回應的大小而不是該請求本身的大小，所以我們的演算法看起來好像是非持續傳送式的而不像傳統的封包排程演算法大部分都是持續傳送式的。我們分別以回應的大小以及窗式控制的機制去仿效 DRR 來決定請求的順序以及其釋放的時間。此順序是依據請求的回應大小以及事先定義讓每個服務類別得到網站吞吐量的比例之服務比重，而釋放時間則是依據網站伺服器的處理速度。在系統評估中，當三個服務類別的服務比重被指定為 6:3:1 時，不論存取網頁的大小為何，此服務品質閘道器可讓它們如預期般的獲得網站總吞吐量的 60%、30%及 10%。並且，擁有最大權重之服務類別得到的網站吞吐量以及使用者感受到的等待時間分別最多改善為沒有差別服務品質時的 176%及 69%。

關鍵字：網站服務品質、差別服務、請求排程

Request Scheduling for Differentiated QoS at Website Gateway

Student: Shuo-Yen Wen Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science
National Chiao Tung University

Abstract

With the explosive growth of Web traffic, the load on a Web server become heavier, leading to the longer user-perceived latency. Website operators would like to employ service differentiation to offer better throughput and user-perceived latency for some specific users. This work presents an HTTP request scheduling algorithm deployed at the website gateway to enable the Web quality of service without modifying any client or server software. Unlike traditional packet scheduling algorithms which are mostly work-conservative, our algorithm appears to be non-work-conservative from the viewpoint of the output link of the gateway because the service time of a request depends on the size of its response, not the size of the request itself. We emulate deficit round robin and the window control mechanism to decide the order and release time of requests, respectively. The order is according to the response size of the requests and the pre-defined service weights, to which the throughput of the service classes will be proportional, whereas the release time is according to the service rate of the Web server. In the evaluation, when the weight ratio 6:3:1 is assigned to three service classes, QoS website gateway lets 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 Quality of Service, Service Differentiation, Request Scheduling

Contents

CHAPTER 1. INTRODUCTION....................1

CHAPTER 2. PROBLEM STATEMENT....................4

2.1 Request Classification....................4

2.2 Request Scheduling....................5

2.3 Server Probing....................7

CHAPTER 3. SYSTEM ARCHITECTURE OF THE QOS WEBSITE GATEWAY....................8

3.1 Architecture of the QoS Website Gateway....................8

3.2 Request Classifier....................8

3.3 Request Scheduler....................11

3.4 Server Prober....................14

CHAPTER 4. IMPLEMENTATION AND EVALUATION....................15

4.1 Implementation....................15

4.2 Evaluation....................16

4.2.1 Evaluation Environment....................16

4.2.2 Evaluation with Fixed-Size Web Pages....................17

4.2.3 Evaluation with Mixed-Size Web Pages....................19

4.2.4 Evaluation with Various Window Sizes....................21

CHAPTER 5. CONCLUSION AND FUTURE WORK....................24

REFERENCES....................26

List of Figures

Fig. 1. Decomposition of Web page download time ... 1
Fig. 2. Keynote business 40 Internet performance index ... 2
Fig. 3. Architecture of the QoS website gateway ... 8
Fig. 4. Syntax of the QoS specification ... 9
Fig. 5. An example of the QoS policy table ... 10
Fig. 6. Concept of the request scheduler ... 11
Fig. 7. Pseudo code of the scheduling algorithm ... 13
Fig. 8. Operation scenario of the implementation ... 15
Fig. 9. Evaluation environment ... 17
Fig. 10. Throughput under various fixed-size Web pages ... 18
Fig. 11. Throughput ratio under various fixed-size Web pages ... 18
Fig. 12. User-perceived latency under various fixed-size Web pages ... 19
Fig. 13. Throughput under various mixed-size Web pages ... 20
Fig. 14. Throughput ratio under various mixed-size Web pages ... 20
Fig. 15. User-perceived latency under various mixed-size Web pages ... 21
Fig. 16. Total throughput under various window sizes ... 22
Fig. 17. Throughput ratio under various window sizes ... 22
Fig. 18. User-perceived latency under various window sizes ... 23

CHAPTER 1. INTRODUCTION

Nowadays, more users connect to the Internet to surf the WWW (World Wide Web). The more accesses to a website, the heavier load will be on this Web server. The busier server leads to the longer user-perceived latency, which means the users will wait for the 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. Fig. 1 shows the time decomposition at browsing a Web page. The browser performs DNS (Domain Name System) lookup to query the IP address of the Web server, establishes a TCP connection, and sends out an HTTP (HyperText Transport Protocol) request. The Web server responds with an HTTP response code in the first packet to indicate the status of the requested page. If the server redirects the request to another server for load balancing, the Web browser will re-initiate DNS lookup, TCP connect, and receive the first packet from the new server. Then the Web browser receives a text-based page and issues new requests to retrieve all other contents, i.e. embedded objects, from the server.

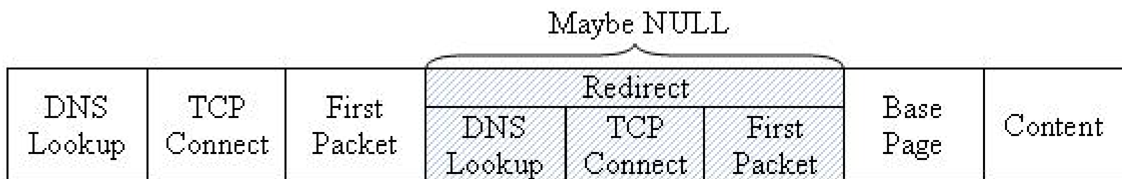


Fig. 1. Decomposition of Web page download time

Here we give an example of the time measurement at downloading a Web page in 2001 from Keynote, a company that measures website performance from its 1,700 measurement computers distributed in 50 metropolitan areas worldwide [1]. Fig. 2 shows the download time for the homepages of 40 important US-based business Web sites [2]. From this figure,

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 QoS (Quality of Service) mechanisms [3, 4], whereas the server bottleneck could be resolved by clustering some 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. RSVP (Resource reSerVation Protocol) [5]. Our research focuses on resolving server bottleneck because website operators can completely control their servers, but cannot do much on improving the whole network performance. The goal is to improve the throughput and reduce the user-perceived latency for some specific users, in other words, to provide service differentiation at the server-side, and thus allows some users to perceive the shorter latency on downloading Web pages.

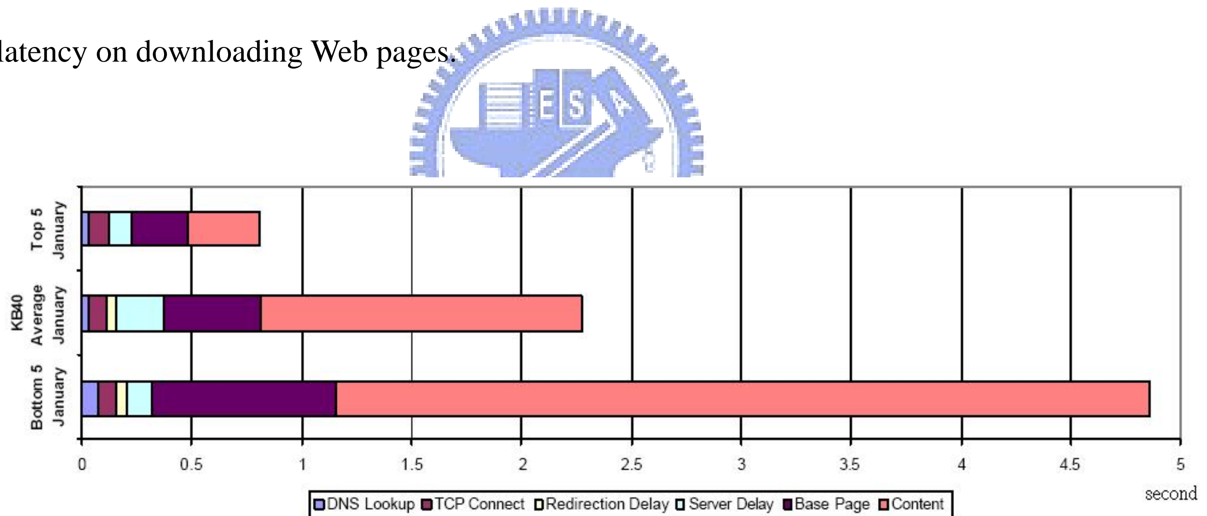


Fig. 2. Keynote business 40 Internet performance index

At the server-side, the HTTP traffic can be controlled at the packet level or the application level. Several recent researches proposed application-level QoS [6-11] 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 [6] or the daemon program [6-11] in the Web server to provide Web QoS. However, the shortcoming is

that those mechanisms are operating systems or server daemons dependent. For the reason of generality, a solution deployed at the website gateway, independent of operating systems and server daemons and transparent to clients and the server, is preferred. Our solution also adopts application-level QoS but differs from other methods in that our approach is deployed at the website gateway. All the HTTP traffic passing through the website gateway will be controlled to provide service differentiation.

Our method is to classify the incoming requests into different service class queues, to schedule the order of each request, and to queue the scheduled requests in the website gateway. The time to release a request depends on the service rate of the server. Unlike traditional packet scheduling algorithms which are mostly work-conservative, our algorithm needs to be non-work-conservative from the viewpoint of the output link of the gateway because the service time of a request depends on the size of its response, not the size of the request itself. Note that the algorithm is work-conservative from the viewpoint of the server and the reverse direction. More precisely speaking, the *order* of transmitting requests from different class queues will result in different shares of server resource, whereas the *time* to release requests will affect the number of concurrent connections between the gateway and the server. In our solution, we emulate DRR (Deficit Round Robin) scheduling [12] and the window control mechanism to decide the order and release time of requests, respectively. In addition, a probing mechanism is used to seize the information, including URL and the response size, of Web pages stored in the Web server to help the DRR scheduling.

The rest of this work is organized as follows. Chapter 2 formally states the problem. Chapter 3 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 our solution are discussed in Chapter 4. Finally, the conclusion and the future work are given in Chapter 5.

CHAPTER 2. PROBLEM STATEMENT

Given a Web server and different classes of clients, the goal is to provide service differentiation by HTTP request scheduling on the website gateway. All HTTP requests originated from clients will pass through the website gateway and the gateway will schedule them to the Web server according to the QoS policies and the service rate of the Web server. The website gateway first classifies the incoming requests into different service classes by inspecting the content of IP headers, HTTP headers, and HTTP payloads. Then the website gateway decides *which* request should be fetched next from different service classes and *when* it should be released to the Web server according to the QoS policies and the service rate of the Web server, respectively. For knowing the characteristics of Web pages stored in the Web server, the website gateway probes the Web server before the on-line operation. In a word, request classification, request scheduling and server probing are the three elements the website gateway does for service differentiation. These issues are discussed more deeply below.

For concentrating on the design of the request scheduling, the dynamically generated Web page and the cluster of servers are not considered in this work. A dynamically generated page varies its URL and response size. This makes the gateway hard to seize the characteristics of dynamic pages. In multiple-server scenarios, the issues of the server load balancing also need to be considered. Therefore, we only focus on a single Web server with static Web pages in this work.

2.1 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 service differentiation. Therefore, the classifier should be content-aware, i.e. see the information contained in the application layer protocols. Below is some usable information for classifying Web requests based on HTTP.

1. URL (Uniform Resource Locator): An URL usually contains host name, port number, username/password, URL path, and file extension.
2. HTTP header: Some useful fields are “Authorization”, “Proxy-Authorization”, and “User-Agent”.
3. HTTP payload: HTTP payload may contain session level identifiers. Cookie and SSL ID (Secure Sockets Layer Identifier) are two examples.

In our work, a content-aware classification mechanism on the website gateway is presented to provide flexible QoS policies for website operators.

2.2 Request Scheduling

After HTTP requests have been classified and accumulated in the corresponding queues, the request scheduler decides *which* request should be fetched next and *when* it should be released to the Web server. For service differentiation, each class queue is assigned a weight and the server resource is proportionally partitioned according to the weights. The larger weight a class has, the more server resource the requests in that class can utilize. In our work, the server resource is partitioned based on throughput because it explicitly stands for the output rate of an HTTP response. Therefore, the request scheduler schedules requests for partitioning the throughput of the server.

Several packet scheduling disciplines, such as PQ (Priority Queuing) [13], WFQ

(Weighted Fair Queuing) [14], WRR (Weighted Round Robin) [15], and DRR [12], can be used for determining which queue to be fetched next. We intend to modify these packet scheduling algorithms for request scheduling. However, unlike traditional packet scheduling algorithms which are mostly work-conservative, our algorithm needs to be non-work-conservative for access link because the service time of a request depends on the size of its response, not the size of the request itself. With PQ, requests are scheduled from the head of a given queue only if all queues with higher priorities are empty. Within each priority queue, requests are scheduled in the FIFO (First In First Out) order. However, if the volume of higher-priority requests becomes excessive, lower-priority requests will be dropped as the buffer allocated to lower-priority queues starts to overflow. With WFQ, WRR, and DRR, each class queue will be assigned a percentage of the server resource according to its weight. However, WFQ implements an O(log N), where N is the number of queues, algorithm that requires selecting the smallest timestamp among all queues. The O(1) WRR provides the correct percentage of the server resource to each class only if requests in the queues have the same response size or the mean of response size is known in advance. Therefore, the O(1) DRR is finally employed as our scheduling algorithm since it is much simpler and it can overcome the limitations of PQ, WFQ, and WRR.

Once the scheduler fetches a request, the next step is to decide its releasing time to the Web server. This is an extra step in request scheduling, compared to packet scheduling. If the releasing rate is too high, the incoming requests will be sent to the server too quick, causing the queues at the gateway to be empty and the requests to be queued on the server and may even overwhelm the server. Note the server itself does not provide any differentiation in processing the requests. Thus, the scheduler should release requests according to the service rate of the Web server. In our work, the window control mechanism is employed to throttle the rate of releasing requests. Combining the concepts of DRR and the window control, the scheduler can partition the throughput of the server and release requests according to the

service rate of the server.

2.3 Server Probing

In order to let the scheduler work accurately to partition the throughput of the Web server, some characteristics of Web pages stored in the server should be known in advance. Without modifying the system kernel or the daemon program in the Web server, *URL* and the *response size* of each Web page on the server are probed from the gateway before the on-line operation or during the system initialization. The probed results are used by the scheduler for the initial accesses of the Web pages, and they will be updated repeatedly by the later accesses of the Web pages, should they get modified during the operation, because the gateway knows the latest response sizes when receiving the Web pages from the server.

CHAPTER 3. SYSTEM ARCHITECTURE OF THE QOS WEBSITE GATEWAY

3.1 Architecture of the QoS Website Gateway

For providing service differentiation, the website gateway performs request classification, request scheduling, and server probing, as discussed in Chapter 2 and shown in Fig. 3. When an HTTP request arrives at the gateway, the request classifier classifies it into a proper class and accumulates it into the corresponding queue according to the QoS policy table. The request scheduler decides *which* request should be fetched next from different service class queues and *when* it should be released to the Web server according to the QoS policy table and the window size, respectively. The server prober probes the characteristics of Web pages stored in the Web server before the on-line operation of the classifier and the scheduler. The probed results are recorded in the Web page table and fed to the request scheduler. The detail of each component is discussed in the following sections.

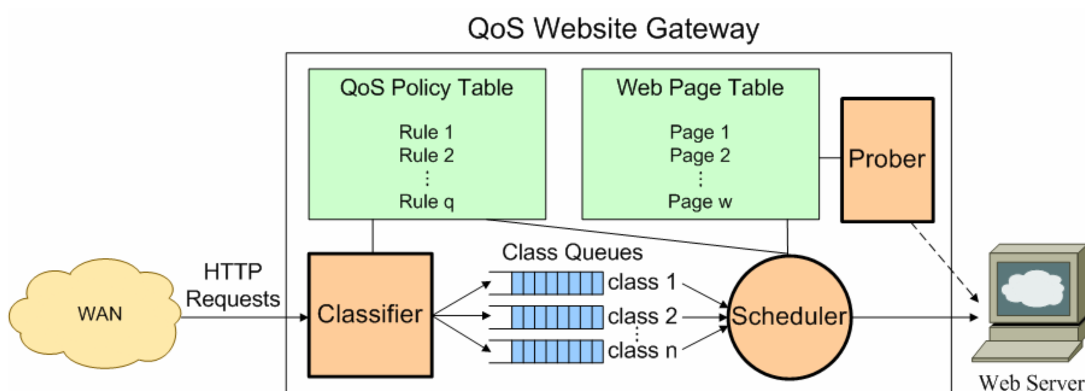


Fig. 3. Architecture of the QoS website gateway

3.2 Request Classifier

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. For a more formalized representation of the policy table, all the rules are written in the XML (Extensible Markup Language) [16] format, as shown in Fig. 4.

```
<?xml version="1.0"?>
<classification>
   <class_id="$class_id" quantum="$quantum"/>
      <$type-1="$pattern-1"/>
      <$type-2="$pattern-2"/>
               ⋮
      <$type-n="$pattern-n"/>
   </class_id>
</classification>
```

Fig. 4. Syntax of the QoS specification

The meaning of each field in Fig. 4 is defined as follows.

- *class_id*: Service class identifier. Each service class has a unique identifier number.
- *quantum*: Expected quantum of the server throughput for a service class. The quantum is used as the weight for the throughput partitioning.
- *type*: Field name inspected from IP packet headers, HTTP headers, or HTTP payloads. The type can be as follows:
  - *src_ip_addr*: IP address of the client.
  - *dst_ip_addr*: IP address of the server.
  - *src_port*: Port number of the client.
  - *dst_port*: Port number of the server.
  - *prot_type*: Protocol type.
  - *src_dn*: Domain name of the client.
  - *dst_dn*: Domain name of the server.

- *url*: Full or partial string of the URL, which possibly includes host name, port number, username/password, URL path, and file extension.
- *user_agent*: Agent of the client, i.e. name of the browser.
- *cookie*: HTTP cookie name.
- *ssl_id*: Secure sockets layer identifier.

- *pattern*: The value corresponded to the *type* field. For example, if the *type* filed is *src_ip_addr*, then the value is the IP address of the client.

A service class may contain multiple rules. Each rule consists of a *type* and a *pattern*. Fig. 5 gives an example of the QoS policy table, in which three service classes are defined.

```
<?xml version="1.0"?>
<classification>
   <class_id="1" quantum="600"/>
      <src_ip_addr="140.113.0.0"/>
      <url="http://foo.com/~golden/"/>
   </class_id >
   <class_id="2" quantum="300"/>
      <src_dn="seed.net.tw"/>
      <cookie="session_id=abcd1234"/>
      <url="http://foo.com/~silver/"/>
   </class_id >
   <class_id="3" quantum="100"/>
      <url=".jpg"/>
      <url=".rm"/>
   </class_id >
</classification>
```

Fig. 5. An example of the QoS policy table

The request classifier compares the information contained in the incoming HTTP 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.

3.3 Request Scheduler

Our scheduling algorithm emulates the deficit round robin to decide which request can be fetched next in order to partition the server resource proportionally according to the response size of the requests and the quanta defined in the QoS policy table. On the other hand, the window control mechanism is employed to throttle the releasing rate so as not to overwhelm the processing capacity of the Web server. The operation of the deficit round robin scheduling and the window control mechanism is shown in Fig. 6.

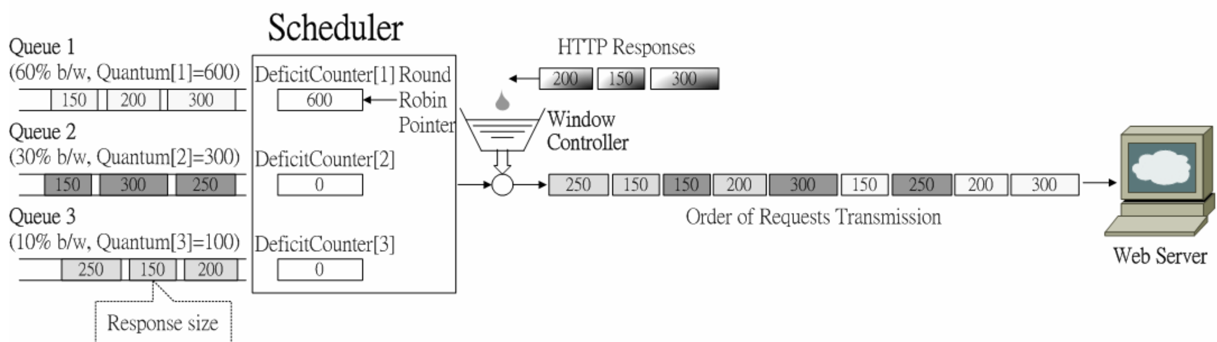


Fig. 6. Concept of the request scheduler

The original DRR scheduling is modified to be capable of handling HTTP requests. The queues store HTTP requests instead of IP packets. The numbers in the blocks in the queues represent the response sizes of the queued HTTP requests, rather than the packet sizes in the original DRR. Each queue has a deficit counter, which tracks the amount of service credits. The quantum of a queue is added to its corresponding deficit counter in a round robin manner. If the value of a deficit counter is larger than or equal to the response size of the request at the head-of-line of the corresponding class queue, the scheduler fetches this request; otherwise, this amount is carried over to the next round. In this way, the server resource, i.e. throughput, can be partitioned proportionally to each service class according to the QoS policies.

The releasing rate of this modified DRR should be throttled such that the released

requests would not overwhelm the processing capacity of the Web server. For this, the window control mechanism is employed to adjust the releasing rate. The window size stands for the number of the maximum concurrent connections between the gateway and the Web server. When a request is released, the window is decreased by one. Conversely, when a resulting HTTP response passes through the gateway, the window is increased by one. The scheduler checks the window before releasing a request. If the window is not empty, the scheduler releases the scheduled request to the Web server; otherwise, it stops scheduling and waits until the window is not empty. In this way, the processing capacity of the Web server can be utilized well without being overwhelmed. A small number is suggested to be assigned to the window size because a large window size may leads to an over-loaded server. However, the effect of the window size is to be studied and discussed in chapter 4.

The pseudo code of the scheduling algorithm is shown in Fig. 7. Initially, all deficit counters are set to zero. Upon arrival of an HTTP 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 *While(TRUE)* loop plays the role of the round robin. The active class queues are processed from the head of the *ActiveList*, say the class *i*. The scheduling algorithm fetches requests from $Queue_i$ when there is enough service quantum and the window is not empty. The service quantum $DeficitCounter_i+Quantum_i$ determines how many requests can be fetched from the $Queue_i$, that is, the sum of the HTTP response sizes of the fetched requests cannot be greater than this service quantum. Before fetching and releasing a request, i.e. invoking $Send(Dequeue(Queue_i))$, the scheduling algorithm checks if the window is not empty. If the window is not empty, the scheduling algorithm releases the request and decreases the window by one. Otherwise, the scheduling algorithm will not release any requests in the $Queue_i$. After a resulting HTTP response has passed through the gateway, the

window will be increased by one.

<Incoming Request>

```
Initialization:
  For (i=0; i<NumofClasses; i=i+1)
    DeficitCounter_i = 0;
Enqueuing module: on arrival of request req
  i = Classify(req);
  If (ExistsInActiveList(i) == FALSE) then
    InsertTailActiveList(i); /* add queue i to active list */
    DeficitCounter_i = 0;
  Engueue(i, req); /* enqueue request req to queue i */
Dequeuing module:
  While (TRUE) do
    If (ActiveList is not empty) then
      i = RemoveHeadActiveList();
      DeficitCounter_i = DeficitCounter_i + Quantum_i;
      While ((DeficitCounter_i > 0) and (Queue_i is not empty)) do
        req = Head(Queue_i); /* get request req from queue i */
        ResponseSize = GetSize(req);
        If (ResponseSize <= DeficitCounter_i) then
          If (Window is not empty) then
            Send(Dequeue(Queue_i));
            DeficitCounter_i = DeficitCounter_i – ResponseSize;
            Decrease Window by 1;
          Else /* return to original condition */
            InsertHeadActiveList(i);
            DeficitCounter_i = DeficitCounter_i – Quantum_i;
            return(); /* exit module */
        Else break; /* skip while loop */
      If (Empty(Queue_i)) then
        DeficitCounter_i = 0;
      Else
        InsertTailActiveList(i);
    Else
      return(); /* exit module */
```

<Outgoing Response>

```
Enqueuing module: on arrival of response rsp
  Engueue(rsp);
Dequeuing module:
  While (TRUE) do
    If (Queue is not empty) then
      Send(Dequeue(Queue));
      Increase Window by 1;
```

Fig. 7. Pseudo code of the scheduling algorithm

3.4 Server Prober

The request scheduler needs the response size of Web pages stored in the Web server when performing scheduling. The server prober is used to probe 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 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 updated repeatedly 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 change; the request scheduler can update the Web page table because it knows the latest response size when receiving this page from the Web server.

CHAPTER 4. IMPLEMENTATION AND EVALUATION

4.1 Implementation

The request classifier and the request scheduler are implemented as a daemon program called "*WebQ*" on the NetBSD [17] 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 Fig. 8. To make the *WebQ* work transparently to both clients and the Web server, the ipnat [18] 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.

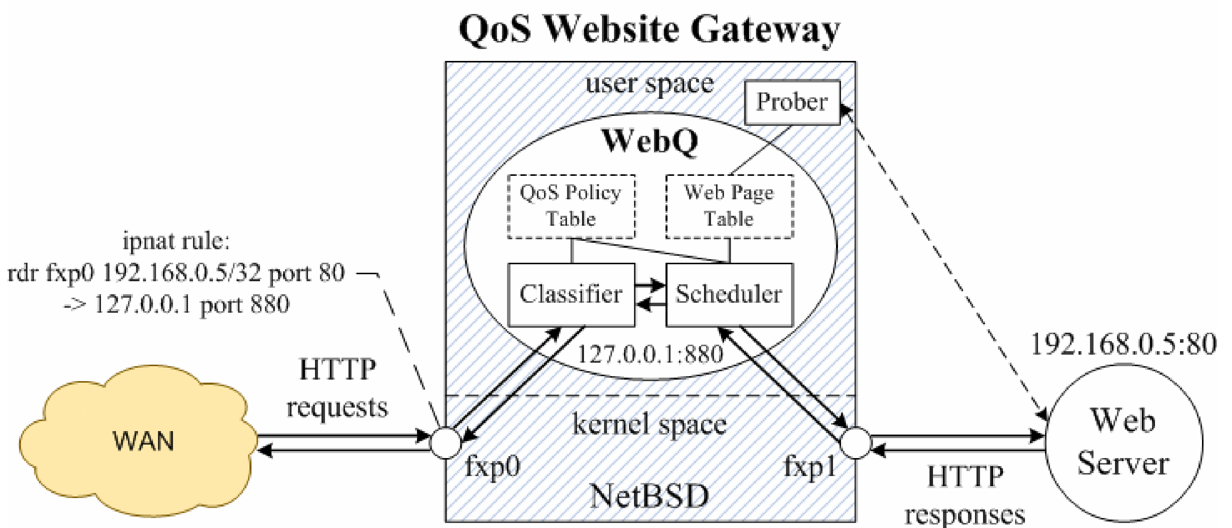


Fig. 8. Operation scenario of the implementation

4.2 Evaluation

The effect of the service differentiation can be evaluated on both the throughput and the 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 our approach. In addition, the effect of the window size is also evaluated.

4.2.1 Evaluation Environment

The evaluation environment consists of an Apache Web server [19], the WebQ gateway, and several computers running the WebBench Web performance testing tool [20], as shown in Fig. 9. The WebBench controller orders the WebBench 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.

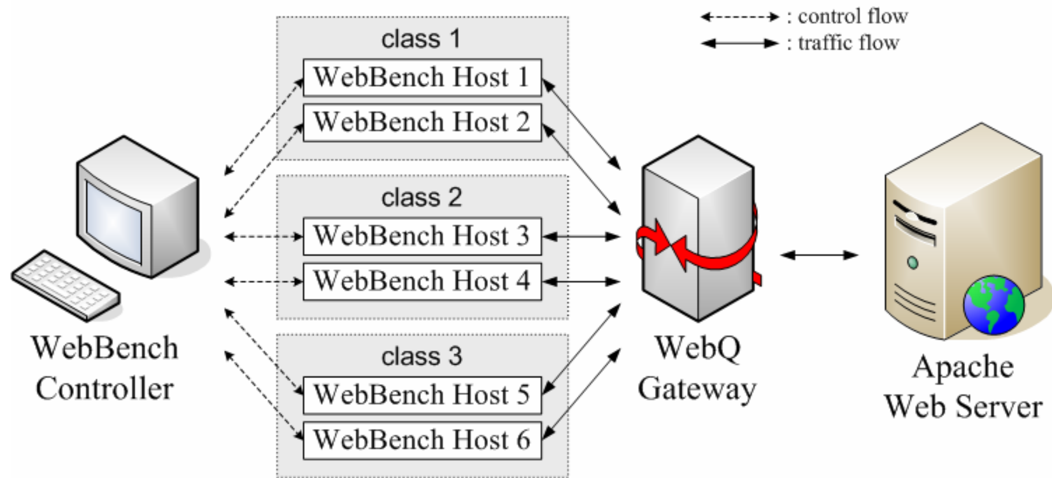
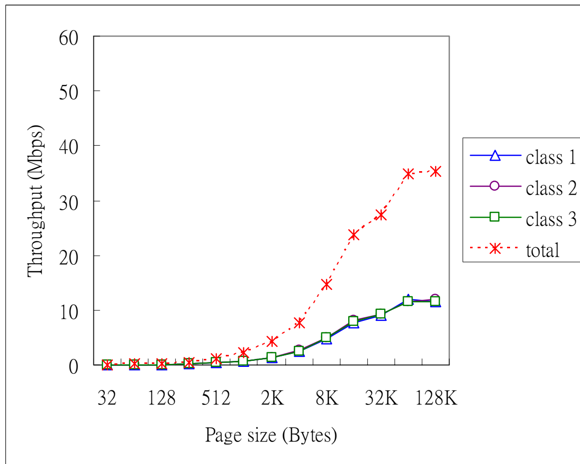


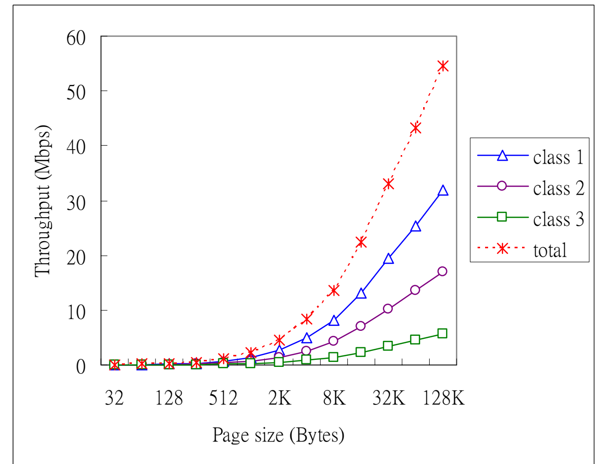
Fig. 9. Evaluation environment

4.2.2 Evaluation with Fixed-Size Web Pages

The evaluation with fixed-size Web pages is to observe the effects of the variation of the page size, which is changed from 32 bytes to 128K bytes. The resulting throughputs are shown in Fig. 10, 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 Fig. 10(a), under the QoS-disabled case, the three service classes get almost the same throughputs because their requests have the same probability of entering the server. Nevertheless, in Fig. 10(b), under the QoS-enabled case, the three service classes get 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 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.



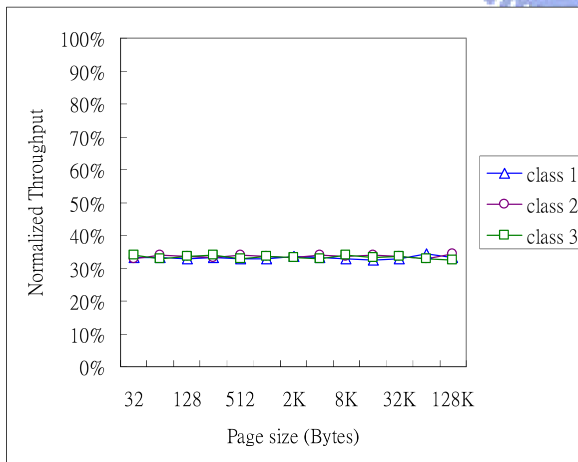
(a) QoS-disabled case



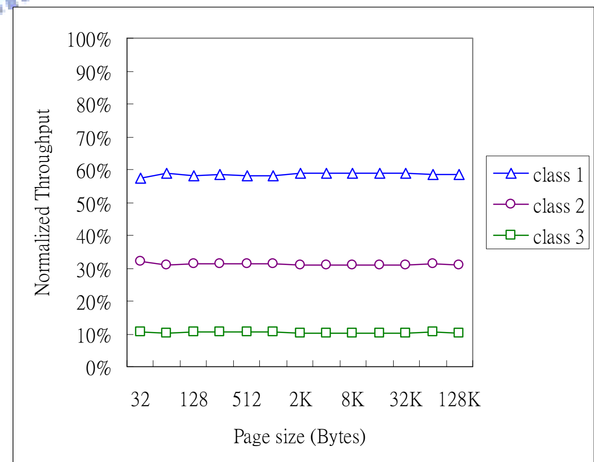
(b) QoS-enabled case

Fig. 10. Throughput under various fixed-size Web pages

For a clearer display of the throughput ratio, the absolute throughputs in Fig. 10 are converted to the throughput ratios shown in Fig. 11. The throughput ratios exhibit 1:1:1 (33.3%:33.3%:33.3%) under the QoS-disabled case, whereas throughput ratios exhibit 6:3:1 (60%:30%:10%) under the QoS-enabled case, regardless whatever page sizes.



(a) QoS-disabled case

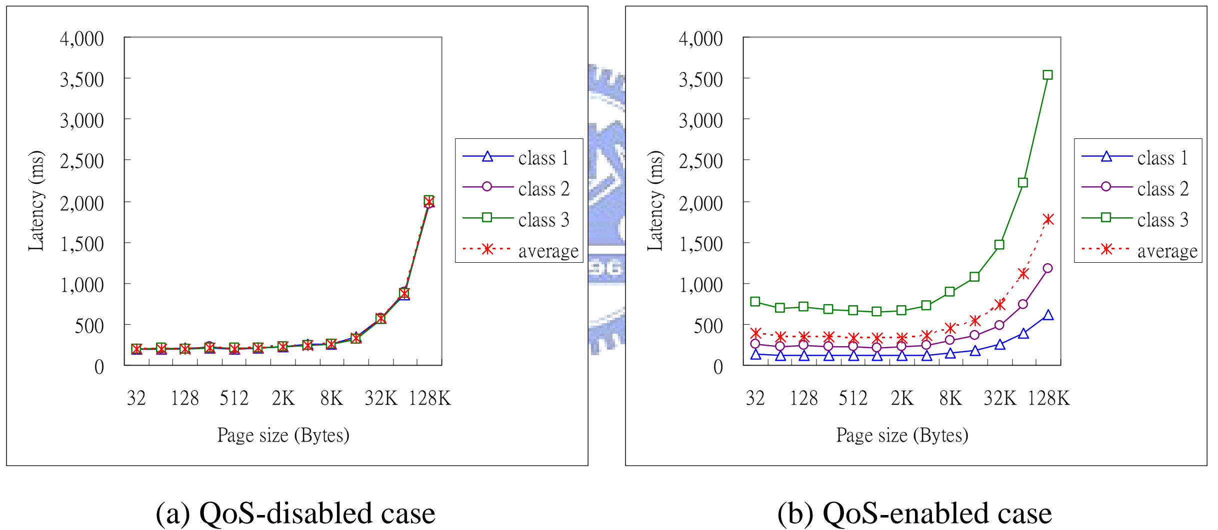


(b) QoS-enabled case

Fig. 11. Throughput ratio under various fixed-size Web pages

The user-perceived latencies under two cases are also compared, as shown in Fig. 12.

The user-perceived latency increases with the page size because the gateway has to process more packets of each response. The three service classes get the same user-perceived latency when QoS is disabled, whereas they perceive different latencies when 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 average user-perceived latency 570 ms under the QoS-enabled case is a little bit longer than 440 ms under the QoS-disabled case because of the queuing delay at the website gateway.



(a) QoS-disabled case (b) QoS-enabled case

Fig. 12. User-perceived latency under various fixed-size Web pages

4.2.3 Evaluation with Mixed-Size Web Pages

In order to evaluate the QoS website gateway under a more realistic environment, the mixed-size Web pages are employed on the server. The page sizes have a lognormal distribution [21], whose probability density function is shown as

$$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-(\ln x - \mu)^2 / 2\sigma^2},$$

where μ (mean for the natural logarithm of the data) and σ (standard deviation for the

natural logarithm of the data) are set to 9.357 and 1.318, respectively.

The absolute throughputs of various classes and their ratios are shown in Fig. 13 and Fig. 14, respectively. The ratios of throughput under the QoS-enabled case still are as expected, close to 6:3:1, demonstrating that our approach works well even in a more realistic environment.

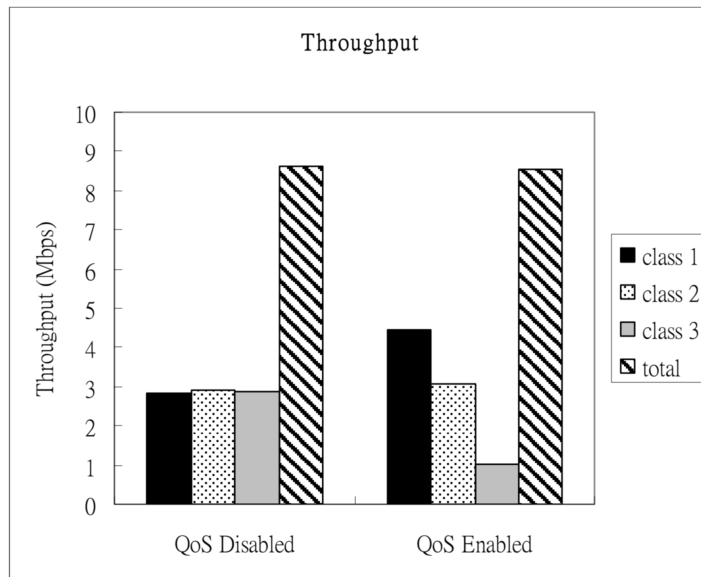


Fig. 13. Throughput under various mixed-size Web pages

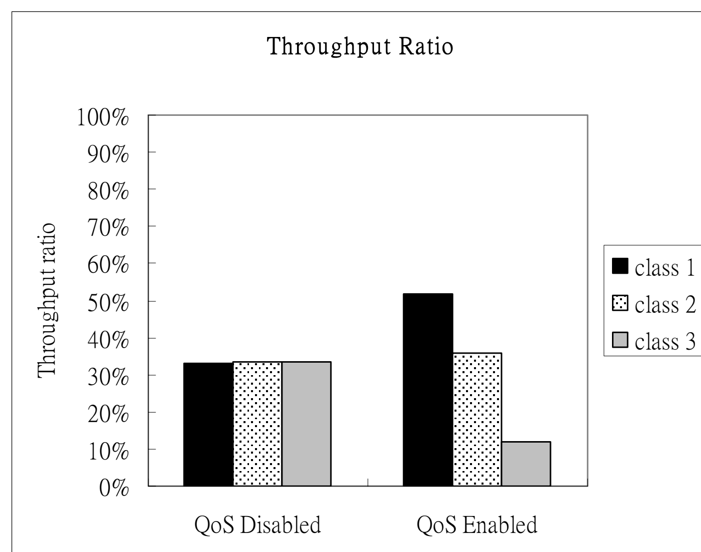


Fig. 14. Throughput ratio under various mixed-size Web pages

The latency is shown in Fig. 15. The observations on the evaluation with mixed-size Web pages are similar to that with fixed-size Web pages.

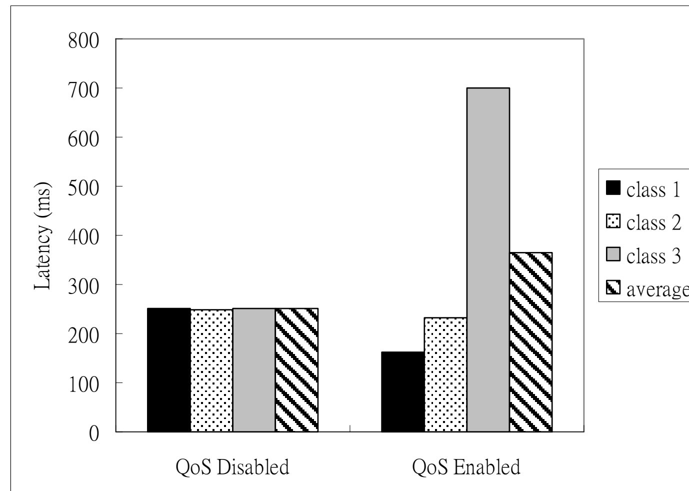


Fig. 15. User-perceived latency under various mixed-size Web pages

4.2.4 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 Fig. 16, 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 our evaluation environment, and furthermore, the aggregated request-sending rate is almost the same under these window sizes.

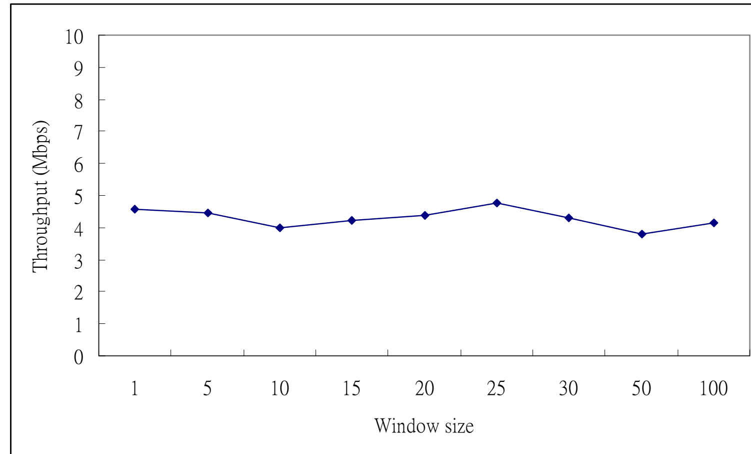


Fig. 16. Total throughput under various window sizes

The effect of the window size on the throughput ratio is shown in Fig. 17, 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 Fig. 18.

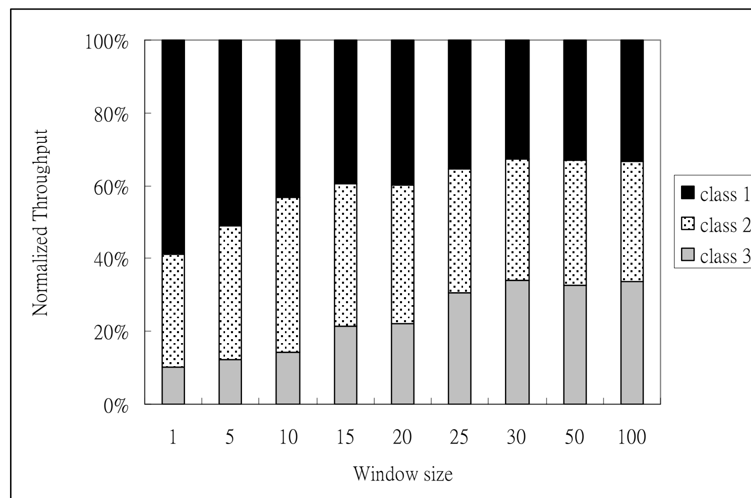


Fig. 17. Throughput ratio under various window sizes

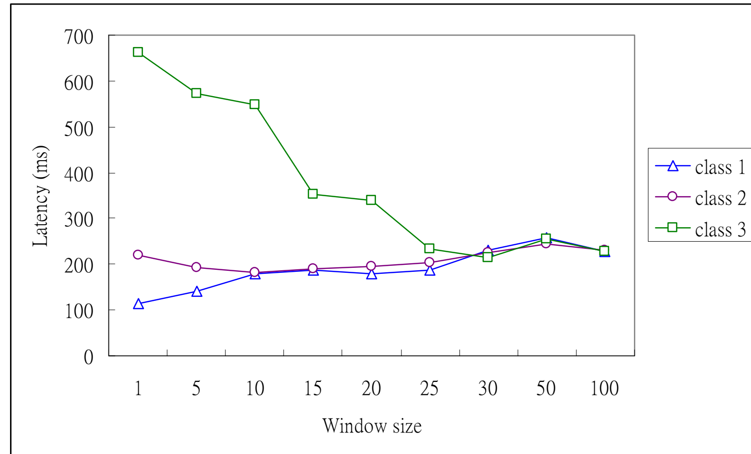


Fig. 18. User-perceived latency under various window sizes

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.

CHAPTER 5. CONCLUSION AND FUTURE WORK

Service differentiation is a way for website operators to provide better throughput and shorter user-perceived latency to some specific users. This work presents a request scheduling algorithm deployed at the website gateway to enable the Web quality of service without modifying client's or server's software. The QoS website gateway consists of a request classifier, a request scheduler, and a server prober. Unlike traditional packet scheduling algorithms which are mostly work-conservative, our algorithm appears to be non-work-conservative for access link; but is work-conservative for the server and the reverse direction because the service time of a request depends on the size of its response, not the size of the request itself. We emulate DRR and the window control mechanism to decide the order and release time of requests, respectively. The order is according to the response size of the requests and the pre-defined service weights. The release time is according to the service rate of the Web server. The HTTP requests incoming to the gateway will be classified and accumulated into the corresponding queues by the content-aware request classifier according to the pre-defined QoS policies. The deficit round robin scheduler with the window control mechanism decides which request should be fetched next and when it should be released to the Web server. The server prober scans URL, 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 with fixed-size Web pages and mixed-size Web pages to demonstrate the robustness of our approach. The throughput and the user-perceived latency of each service class are measured to demonstrate the effect of the service differentiation. The evaluation is performed with three service classes whose weight ratio is set to 6:3:1, and various Web pages whose sizes are ranging from 32 bytes to 128K bytes.

Under the QoS-disabled case, the three service classes get almost the same throughputs and the user-perceived latencies (1:1:1). On the contrary, under the QoS-enabled case, the three service classes get expected throughputs and user-perceived latencies as the pre-defined service weights (6:3:1), regardless whatever page sizes. In our evaluation, the throughput of the class with the largest weight is improved by up to 176%, while that of the class with the smallest weight is penalized by 52%. On the other hand, the user-perceived latency of the former is improved by up to 69%, while that of the latter is penalized by 75%.

There are mainly two directions for future works. The Web pages on a Web server can be static or dynamically generated. The URL and the response size of a static page are easy to seize and are used for the scheduling. A dynamically generated Web page varies its URL and response size. When dealing with dynamical Web pages, the QoS website gateway has to be equipped with a more sophisticated server prober to correctly estimate the response size. Furthermore, enabling service differentiation at the QoS website gateway for a cluster of servers is also a considerable work. In this case, the QoS website gateway has to schedule the requests and balance the server load simultaneously.

REFERENCES

[1] Keynote Systems Inc., http://www.keynote.com

[2] P. Mills and C. Loosley, "A performance Analysis of 40 e-Business Web Sites," *CMG J. of Computer Resource Management*, issue 102, 2001.

[3] R. Braden, D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture: an Overview," RFC 1633, June 1994.

[4] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Service," RFC 2475, Dec. 1998.

[5] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification," RFC 2205, Sep. 1997.

[6] J. Almeida, M. Dabu, A. Manikutty, and P. Cao, "Providing Differentiated Levels of Service in Web Content Hosting," in *Workshop Internet Server Performance*, Madison, WI, June 1998.

[7] R. Pandey, J. F. Barnes, and R. Olsson, "Supporting Quality of Service in HTTP Servers," in *Proc. 7th Annu. ACM Symp. Principles of Distributed Computing*, Puerto Vallarta, Mexico, June 1998, pp. 247-256.

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

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

[10] 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, pp. 22-28, Sep. 2000.

[11] X. Chen and P. Mohapatra, "Performance Evaluation of Service Differentiating Internet

Servers," *IEEE Trans. Computers*, vol. 51, issue 11, pp. 1368-1375, Nov. 2002.

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

[13] C. Semeria, "Supporting Differentiated Service Classes: Queue Scheduling Disciplines," Juniper Networks Inc., Sunnyvale, CA, Part Number: 200020-001, Dec. 2001.

[14] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in *Proc. ACM SIGCOMM Computer Communication Review*, vol. 19, issue 4, Austin, TX, Aug. 1989, pp. 1-12.

[15] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, "Weighted Round-Robin Cell Multiplexing in a General-Purpose ATM Switch Chip," *IEEE J. Selected Areas in Communications*, vol. 9, issue 8, pp. 1265-1279, Oct. 1991.

[16] M. Murata, S. St. Laurent, and D. Kohn, "XML Media Types," RFC 3023, Jan. 2001.

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

[18] IP Filter - TCP-IP Firewall-NAT Software, http://coombs.anu.edu.au/~avalon

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

[20] WebBench, http://www.veritest.com/benchmarks/webbench

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