國立交通大學

資訊科學系

碩士論文

應用公平佇列方法

在網路通道閘道器上之請求排程

Request Scheduling

with the Fair Queuing Discipline at Access Gateway

研 究 生：曹樂淇

指導教授：林盈達 教授

中華民國九十三年六月

應用公平佇列方法在網路通道閘道器上之請求排程

Scheduling Requests
with the Fair Queuing Discipline at Access Gateway

研 究 生：曹樂淇　　　　　Student：Le-Chi Tsao

指導教授：林盈達　　　　　Advisor：Dr. Ying-Dar Lin

國 立 交 通 大 學

資 訊 科 學系

碩 士 論 文

A Thesis

Submitted to Institute of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

應用公平佇列方法
在網路通道閘道器上之請求排程

學生：曹樂淇　　　　　　　　　　　指導教授：林盈達

國立交通大學資訊科學研究所

摘要

對 ISP 的客戶而言，當對外的連線網路成為網路瓶頸時，最常見的因應之道是在使用者端的閘道器上採用公平佇列的方法。然而，當網路同時有過多的下載行為時，會使得對外連線網路的下行鏈結成為瓶頸，此時在使用者端的閘道器上使用公平佇列並不能解決此一問題。這是因為下載的回應是在 ISP 端的閘道器上形成佇列，而不是在使用者端的閘道器上形成佇列。針對這樣子的情形，可以對使用者端閘道器上的請求佇列做排程來管理 ISP 端閘道器上的回應佇列。故本論文先陳述二個使用公平佇列方法來實踐請求排程時會遇到的問題，分別是釋放請求的時機以及順序。而後提出一個基於公平佇列方法的請求排程，其中包含了請求型式的公平佇列與視窗服務速率控制器。前者藉著依從一般化的行程共享精神來達成加權的頻寬使用比例與頻寬共享。後者藉由控制共存的回應數量來達成高頻寬使用率並且降低回應延遲時間。透過模擬與實驗的結果，顯示出各個類別間的公平指數分別為 0.89 與 0.87，回應延遲時間則是分別降低了 23.44%與 30%。此外，同樣是讓頻寬使用率達到滿載，對中央處理器的負擔而言，得到控制的共存回應數量會比無限制的回應數量要來得少，如此大約可省下 1/4 的處理耗費。

關鍵字： *請求排程、網路通道閘道器、公平佇列*

Request Scheduling
with the Fair Queuing Discipline at Access Gateway

Student: Le-Chi Tsao **Advisor: Dr. Ying-Dar Lin**

Department of Computer and Information Science

National Chiao Tung University

Abstract

For ISP's customers, when the access link becomes the bottleneck to connect with the Internet, a fair queuing (FQ) discipline deployed at a user-side access gateway is the present solution. However, deploying a FQ discipline at a user-side gateway cannot handle the case when the downlink is the bottleneck, which may results from exceeding concurrent responses in downloading. In the case, responses are queued at the ISP-side, not the user-side, gateway. A solution is scheduling the requests at the user-side gateway to manage the responses queued in the ISP-side gateway. This works first reveal that scheduling requests relying on the fair queuing discipline has two problems in terms of the sending timing and the ordering of requests. Next, we propose a fair-queuing based request scheduling (FQRS) method, consisting of a request-based fair queuing (RFQ) discipline and a window-based service-rate control (WRC) mechanism. RFQ approximates the generalized processor sharing (GPS) model to provide weighted fairness and bandwidth sharing between classes. WRC controls the number of concurrent responses over the downlink to provide almost full bandwidth utilization and reduces the user-perceived latency. The results in simulation and field trial demonstrate FQRS provides short-term fairness in 0.89 and 0.87 respectively while reduces 23.44% and 30% of user-perceived latency separately. Besides, FQRS saves 1/4 of CPU loading with the same throughput comparing to the one no FQRS is implemented.

Keywords: request scheduling, access gateway, fair queuing

Contents

CHAPTER 1 INTRODUCTION **1**

CHAPTER 2 SCHEDULING REQUESTS WITH FAIR QUEUING DISCIPLINES **3**

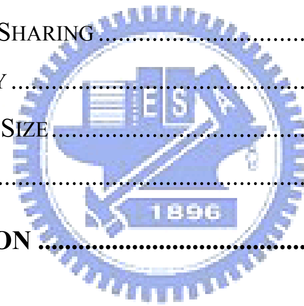
2.1 THE TIMING OF RELEASING REQUESTS 3
2.2 THE DETERMINATION OF THE NEXT REQUEST 4

CHAPTER 3 FAIR-QUEUING BASED REQUEST SCHEDULING DISCIPLINE **7**

3.1 OVERVIEW OF FQRS 7
3.2 REQUEST-BASED FAIR QUEUING (RFQ) 8
3.3 WINDOW-BASED SERVICE RATE CONTROLLER (WRC) 11
3.4 EXAMPLES 13
3.5 DISCUSSIONS FOR EXCEPTIVE CASES 15

CHAPTER 4 SIMULATION RESULTS **18**

4.1 TOPOLOGY 18
4.2 WEIGHTED FAIRNESS AND SHARING 19
4.3 LOWER AVERAGE LATENCY 20
4.4 ADJUSTMENT OF WINDOW SIZE 22
4.5 CHOICE OF F AND U_{MAX} 23



CHAPTER 5 IMPLEMENTATION **25**

5.1 ARCHITECTURE 25
5.2 FIELD TRIAL 27

CHAPTER 6 CONCLUSIONS AND FUTURE WORKS **30**

REFERENCES **32**

List of Figures

FIG.1 THE TYPICAL TOPOLOGY TO ACCESS THE INTERNET AND THE INTERNAL ARCHITECTURE OF FAIR-QUEUING BASED REQUEST SCHEDULING (FQRS). 7

FIG. 1 PROCEDURES OF THE *RFQ* 9

FIG. 3 PROCEDURE OF WINDOW-BASED SERVICE-RATE CONTROLLER 13

FIG. 4 EXAMPLE FOR FQRS IN HANDLING BACKLOGGED TRAFFIC. 14

FIG. 5 EXAMPLE FOR FQRS IN HANDLING REQUESTS WHICH MEET AN EMPTY CLASS QUEUE. THE ACC_D IS RESET AS THE MINIMUM AMONG ALL OTHER ACC COUNTERS 15

FIG. 6 TWO POTENTIAL INTEGRATED ARCHITECTURES FOR HANDLING THE NETWORK WHEN EXCEPTIVE TRAFFIC COEXISTS 17

FIG. 7 SIMULATION TOPOLOGY FOR THREE CLASSES WITH SERVICE RATIO 4:2:1 18

FIG. 8 BANDWIDTH USAGE OF THREE CLASSES WITH SERVICE RATIO 4:2:1 IN FOUR PHASES 20

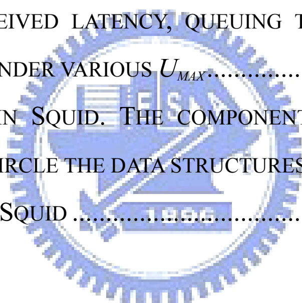
FIG. 9 USER-PERCEIVED LATENCY COMPARISON BY DECOMPOSING TIME FACTORS: QUEUING TIME AND TRANSMISSION TIME 21

FIG. 10 THE SIZE OF W_{MAX} IS STEADY IN CASE OF INSUFFICIENT TRAFFIC. 23

FIG. 11 VARIATIONS OF USER-PERCEIVED LATENCY, QUEUING TIME, AND THE NUMBER OF PACKETS QUEUED IN ISP-SIDE ROUTER UNDER VARIOUS U_{MAX} 24

FIG. 12 THE TRANSACTION FLOW IN SQUID. THE COMPONENTS OF FQRS ARE SHAPE IN DOUBLE RECTANGLES. DOTTED RINGS CIRCLE THE DATA STRUCTURES 26

FIG. 13 TEST BED FOR FIELD TRIAL IN SQUID 27



List of Tables

TABLE 1 THE FUNCTIONS OF TRANSACTION FLOW FUNCTIONS IN SQUID 26
TABLE 2 USER-PERCEIVED LATENCY COMPARISONS. FQRS SQUID SPENDS ADDITIONAL QUEUING TIME. 28
TABLE 3 COMPARISON BETWEEN FQRS AND THE ORIGINAL SQUID IN PERCENTAGE OF CPU TIME.......... 28

Chapter 1 Introduction

Currently, numerous enterprises connect to the Internet to exchange data by the access link of ISP. Generally speaking, ISPs are willing to invest money in expanding the backbone bandwidth to provide their customers better service. However, to minimize costs, their customers often delay upgrading the bandwidth of the access link, causing it becomes the potential bottleneck to access the Internet.

For enterprise users, to provide differentiated service for important connections through the access link, the common solution is scheduling packets with fair queuing disciplines at their access gateway. Unfortunately, the solution fails when the downlink is the bottleneck. In this case, packets are queued at the ISP-side edge router, not at the user-side gateway, for traversing the bottlenecked access link. Scheduling packets at a user-side access gateway is useless because the packets have passed the bottleneck.

This work aims to demonstrate that scheduling requests, instead of packets, at the access gateway can solve the mentioned failed case of packet scheduling. The bandwidth of downlink can be managed by controlling the releasing of uplink requests. Such a solution is based on that most applications running over the Internet are client-server model, i.e. request/response model, such as HTTP, FTP, and E-mail. Request scheduling is used in some recent studies to provide Web QoS [4]. These studies provided QoS services on a single Web server [5-7], caching proxies [9, 10] and server farms [11-13]. No published studies discussed how to design request scheduling at the access gateway. Two of our previous works, Web BM [14] and RQS [15], have targeted the similar object. However, both of them may be hard to implement due to their complexities.

First, we identify two problems occurring in scheduling request with the fair queuing disciplines. One concerns the timing of releasing requests and the other concerns the determination of the next released request. Subsequently, we propose a Fair-Queuing based Request Scheduling (FQRS) method, consisting of a Request-based Fair Queuing (RFQ) discipline and a window-based service-rate controler (WRC). The RFQ discipline provides *weighted fairness* and *bandwidth sharing* as other FQ disciplines since it is also based on the Generalized Processor Sharing model [1]. The WRC keeps the downlink at high utilization, but not in congestion. The WRC achieves the goal by monitoring the utilization and controlling the number of *outstanding* responses. A response is denoted as outstanding if its corresponding request is sent out, but the response has not been fully received.

The fairness provided by FQRS may degrade with the increase of the arrival traffic not triggered by custom side requests. The other challenging case is that receiving an extreme-long response in the fixed rate equal to the downlink bandwidth, where the fixed rate means that the response is carried by non-congestion-aware protocol, like UDP. Obviously, except the malicious attacks, this challenging case does not occur in the Internet. Further discussions are given in the end of Chapter 3.

The remainder of the paper is organized as follows. Chapter 2 identifies the two problems occurring in scheduling requests with the fair queuing discipline. Also, the ideas in our previous works, Web BM and RQS, for handling these problems are briefed. Chapter 3 proposes the FQRS algorithm, consisting of the RFQ and the WRC, and discusses its potential fairness degradation in two cases. Chapter 4 verifies our algorithm on providing weighted fairness and bandwidth sharing while shortening the transmission time of responses. Chapter 5 demonstrates the effect of the FQRS through field trail, where the FQRS is implemented in Squid [16], an open-source web proxy package. Chapter 6 gives the conclusion and future works.

Chapter 2 Scheduling Requests with Fair Queuing Disciplines

FQ disciplines approximate the GPS model can schedule packets well. However, as mentioned below, two problems may occur in request scheduling by FQ disciplines. The user traffic handled in a request scheduling includes the uplink requests and downlink responses, not just the downlink packets.

2.1 The Timing of Releasing Requests

2.1.1 Problem Statement

The FQ disciplines send the next packet when the last packet has been transmitted. The bandwidth of a bottlenecked link is totally consumed by the scheduled packets themselves. That is, the packet transmission *monopolizes* the link bandwidth. Therefore, once a packet is selected from a queue by a scheduling discipline, it is transmitted in the rate equal to the capacity of this link.

However, the request scheduling *cannot* send the next request following the last request right away. The bandwidth of bottlenecked downlink is consumed by the responses, rather than scheduled requests. Sending requests one-by-one results in too heavy concurrent outstanding responses because the response is usually much larger than the request and a response transmission does not monopolize the downlink bandwidth. Too many concurrent outstanding responses cause the downlink in a serious congested condition.

On the other hand, it is not appropriate in request scheduling that no request is allowed to be sent until the preceding request gets its response completely. When a request is scheduled, the bandwidth of the bottlenecked downlink is not consumed

immediately until the corresponding response returns. Obviously, sending the next request until receiving the whole preceding response would waste the bandwidth of downlink, causing low bandwidth utilization.

2.1.2 Potential Solution

Obviously, appropriate outstanding responses are necessary in scheduling request with FQ disciplines. The request scheduler WebBM, our previous work, allocates the bandwidth among responses precisely to avoid the link from waste and congestion. For not wasting, WebBM releases next request in advance so that its response can exploit the idle bandwidth as soon as the preceding response finishes. For not congestion, WebBM only releases the request whose bandwidth requirement of response fits the coming idle bandwidth.

To forecast when and how many idle bandwidth there is, WebBM needs the information about the rate and the finish time of each response transmission. To decide which request is the next and its advanced releasing time, WebBM needs the potential bandwidth requirement, and the time between releasing a request and receiving the first packet of its corresponding response. Although the required information can be abtained from historical statistics, inaccurate information may affect the access link utilization. Besides, no historical data exists in the initial state, and thus WebBM may encounter great difficulties in an open environment.

2.2 The Determination of the Next Request

2.2.1 Problem Statement

Due to the rule of approximating GPS, the FQ disciplines tend to select the packet that would complete service first in the fluid GPS model as the next packet. Since packets are transmitted in a full capacity of link, the calculation of the potential service completion time only involves two known parameters, packet arrival time and

packet size. Thus, the potential service completion time can be easily calculated when a packet arrives. For two packets arriving at the same time, the packet size decides the order of service completion time. A smaller packet finishes service earlier.

However, for request scheduling, the service completion time of a transaction is unknown. The response size of a transaction is unknown until the first packet of the returning response is got. At that time, the size of a response can be extracted from the application header of this packet. Moreover, a response may not be transmitted in a constant rate. For this reason, request scheduling cannot serve requests simply by the order of service completion time to provide fairness. Hence, the determination of which request will be scheduled as the next is the second problem.

2.2.2 Potential Solution

The RQS [15], another our previous work, had the same four goals as this work: proportional fairness, bandwidth sharing, congestion reduction, and fully link utilization. It merged the algorithm from WebBM to estimate the timing of releasing next request and a selection discipline derived from the Deficit Round Robin (DRR) [17], which is one of packet-based fair queuing algorithms. DRR provides fairness by proportional quantum in bytes among classes. Class queues are served in a round-robin manner. The *quantum* of each class's deficit counter stands for the allowed service amount, i.e. the total length of packets can be *transmitted* in each round.

RQS measures fairness by counting the size of received responses size. Thus, the quantum of one class stands for the size of responses allowed to be *received* in one round. When planning to send a request, RQS selects the first request from a queue which is in turn meanwhile owns sufficient quantum for the given corresponding response size. If the quantum is insufficient, the deficit counter of this queue saves the remaining quantum for use in the next round. The scheduler then serves the next class

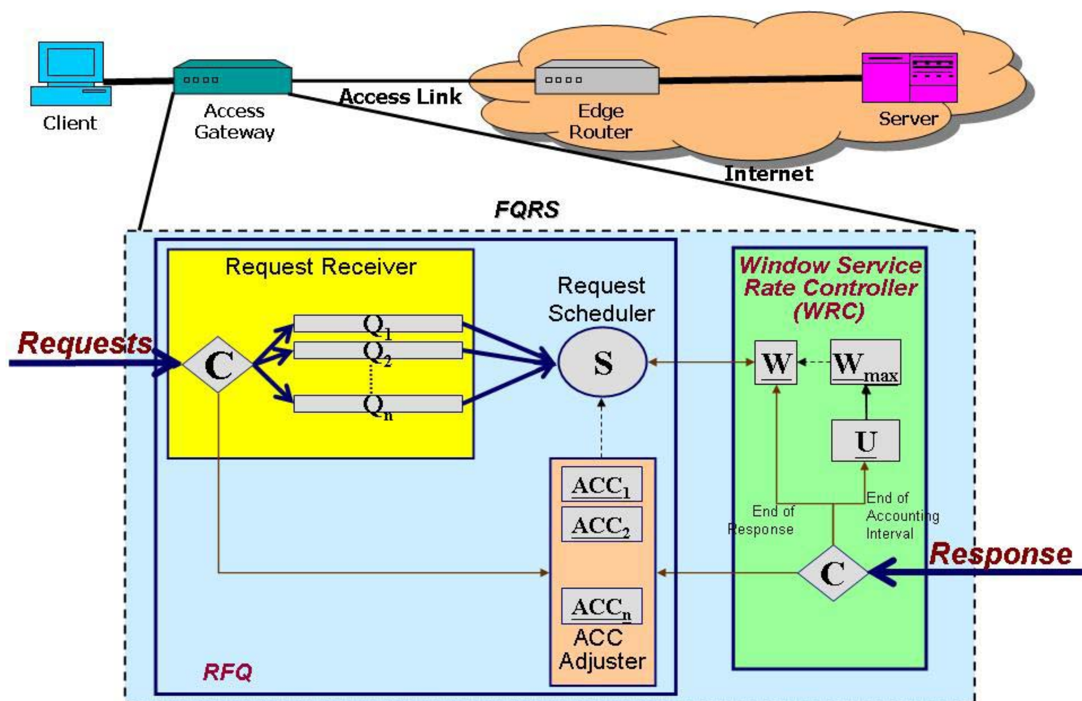
queue.

The selection discipline used in RQS is designed under the assumption that the size of a response is always given as the request arrives. Inaccurate response size may degrade the fairness between classes. Actually, the response size is unknown when the request arrives, and it is extracted until the first response packet is received in most case. For this problem, RQS has a compensation mechanism to prevent the unfairness resulted from inaccurate estimation of response size at scheduling. Once the obtained real size is different from the estimated one, the corresponding deficit counter is compensated with the difference in size in that round.

Chapter 3 Fair-Queuing based Request Scheduling Discipline

The chapter proposes a Fair-Queuing based Request Scheduling (FQRS) discipline. It is expected to provide proportional fairness, sharing resource, full bandwidth utilization, and short transmission latency.

3.1 Overview of FQRS



A — A is a variable used in FQRS

⟶ Transaction flow

A ⟶ B — A triggers B to refresh or to take an action

A ⇢ B — A is referenced by B

Fig.1 The typical topology to access the Internet and the internal architecture of fair-queuing based request scheduling (FQRS).

Figure 1 is a typical network topology that an enterprise accesses the Internet services. Requests sent from clients go through the access gateway and the uplink of the access link to remote servers, and the corresponding responses answered by the

remote servers return to clients through the downlink of the access link and the access gateway. The FQRS discipline is deployed at the user-side access gateway. The FQRS consists of request-based fair queuing (RFQ) and window-based service-rate controller (WRC). The former decides which request is the next one while the latter determines the timing to release requests. The request scheduler in RFQ makes the decision by referring the ACC counters. The ACC adjuster maintains the ACC counters and the request receiver in RFQ executes the enqueueing process when requests arrive**.**

3.2 Request-based Fair Queuing (RFQ)

3.2.1 ACC Counter and Next Request Selection

Each ACC counter accumulates the received normalized service of each class. Assume a class receives S bytes responses from beginning. Then, the normalized service of this class is the value which is S divided by the weight of this class.

Every time invoked by WRC, the *request scheduler* sends out the HOL (head-of-line) request from the class which has the minimum ACC counter. Under the assumption that all classes have backlogged requests, a class with smaller ACC value represents that it received less service than other classes at present. Thus, it is reasonable that selecting a request from this class will achieve the fairness between classes.

On the other hand, to avoid one class from occupying all resource after a long idle period, its ACC counter is updated to the minimal values among other ACC counters when a request arrives. Without the update, the ACC counter of this class may be far smaller than other ACC counters, causing that no request can be selected from other classes for a long time. Basically, the RFQ follows the concept in the fair queuing service disciplines that the class using the idle bandwidth should not be

punished.

3.2.2 Basic Procedures

```
Scheduler
if (Q_no = GetMinAccQ() != Null) {
    SendHeadReq(Q_no)
    W++
}
```

(a) Procedure of *request scheduler*

```
AccAdj
if (TypeRsp)
     ACC_Qno += len / WT_Qno
if (TypeReq)
     ACC_Qno = GetMinAcc()
```

(b) Procedure of *ACC Adjuster*

```
ReqRcv
req = Rcv()
Q_no = Classfy(req)
if ( Empty(Q_no) )
    AccAdj(TypeReq, Q_no)
ENQUEUE(req)
if (W == 0)
    Scheduler()
```

(c) Procedure of *request receiver*

Fig. 1 Procedures of the *RFQ*

Fig.2 (a), (b), and (c) list the pseudo codes of the three components in RFQ. The *request scheduler* picks the class queue with the minimum ACC counter and sends the HOL request in that queue out. After that, the window size is increased by one to denote one more outstanding response.

As shown in Fig. 1, the *ACC adjuster*, pointed by two arrows, represents that it is invoked in two cases. First, it is invoked when WRC receives a response of any class. The length of the response would be added to the ACC counter of the corresponding class. Second, it is invoked when traffic is infused to an empty class queue. In the case, the ACC counter would be reset as the minimum value among other ACC counters.

The *request receiver* classifies and enqueues all incoming requests. If the arrival request is classified to an empty queue, the *request receiver* informs the *ACC adjuster* to reset the ACC counter of this idle class. If the system is idle, that is, no responses are outstanding, the *request receiver* actively asks the *request scheduler* to release the coming request immediately. The variable *W* would be introduced in section 3.3.

3.2.3 Mapping to Packet-based GPS

In packet-based fair queuing disciplines, each packet is tagged with timestamps. Let T_f^i be the finish timestamp of the ith packet of flow *f*, V_f^i be the virtual time when it arrives, and L_f^i and ϕ_f denote its length and the weight of flow *f*, respectively. T_f^i is defined as follows:

$$T_f^i = \max\left\{T_f^{i-1}, V_f^i\right\} + \frac{L_f^i}{\phi_f} \tag{1}$$

where $T_f^0 = 0$ and $i \geq 0$. Several maintenance of V_f^i were discussed, such as that in SCFQ [2] and SFQ [3].

In the RFQ, only the HOL request of each queue is tagged with accumulated received normalized response size of that class as timestamp when responses or requests are received. Let ACC_k^i be the accumulated received normalized response size for class *k* before the (i+1)th request is released. The virtual time V_k^i herein is defined as the minimum value of all other classes' ACC counters meanwhile the request meets an empty class queue when it arrives, or else it is defined as zero. Suppose the number of classes is *m*,

$$ACC_k^i = \max\left\{ V_k^i, ACC_k^{i-1} \right\} \tag{2}$$

$$V_k^i = \begin{cases} \min\left\{ ACC_1^i, \cdots, ACC_{k-1}^i, ACC_{k+1}^i, \cdots, ACC_m^i \right\}, & \textit{if reviving case} \\ 0 & \textit{else} \end{cases} \tag{3}$$

In packet-based fair queuing disciplines, the HOL packet of a queue with minimum finish timestamp is selected as the next. It means that the packet could complete service soon can be served first. In other words, the one asking least resource gains service before others. In FQR, the HOL request of a class with minimum accumulated received response size is served as the next. It means the class having consumed least resource can be served earlier.

3.3 Window-based Service Rate Controller (WRC)

The WRC component adjusts the number of concurrent outstanding responses by measuring the bandwidth utilization of the link, denoted by U. As shown in Fig. 1, the variable W is used to record the number of outstanding responses at present. The variable W_{max} represents the maximum number of outstanding responses allowed by WRC.

3.3.1. The Measurement of Utilization

The utilization U is acquired through dividing the received response in bytes by the duration of *recomputing interval*. How to define the interval is a major issue. Defining a fixed interval is simple, but the value of the interval is hard to given. Too short interval may not reflect the average of U and may lead W_{max} to be adjusted frequently. On the contrary, too long interval may lead the link into low utilization since W_{max} is changed slowly and cannot immediately reflect the low utilization.

Compared to the fixed interval, this work computes U in a variant interval. The interval is defined as the period that $F \times W_{max}$ ending events of response transmission occur, where F is a constant and its value is not sensitive for the resulted utilization, as shown in the simulation later. By such a definition, the *recomputing interval* is dependent with the response size and the maximum number of concurrent outstanding responses.

3.3.2 Window-based Outstanding-response Control

WRC uses a window-based mechanism to adjust the number of outstanding responses. When U is lower than the expected utilization, denoted as U_{max}, W_{max} would be increased to release more requests so that more concurrent outstanding responses can exploit the bandwidth and then raise the utilization U. On the contrary, when U is higher than U_{max}, W_{max} would be decreased so that fewer outstanding responses compete for the bandwidth.

Assume that WRC has backlogged requests. The adjusting rule used in WRC is simple and shown as follows.

$$\begin{cases} W_{\max}^{i+1} = W_{\max}^{i} + 1 & (\ U^{i} < U_{\max}\ ) \\ W_{\max}^{i+1} = W_{\max}^{i} - 1 & (\ U^{i} > U_{\max}\ ) \end{cases} \quad (4)$$

where $W_{\max}^{i}$ is the maximum concurrent outstanding responses allowed in the i-th interval and $W_{\max}^{0} = 1$ while U^{i} represents the bandwidth utilization computed in the i-th interval. Notably, the rules only applied when W=W_{max}. When W<W_{max}, it is wrong to expect the raise of U by increasing W_{max}.

In order to quickly boost U at beginning, a fast-start mechanism is used on adjusting W_{max}. W_{max} is doubled every interval when U is less than half of U_{max}. Until U is greater than half of U_{max}, WRC begins to use the above rule.

3.3.3. Basic Procedure

The pseudo code of the WRC is listed in Fig. 3. When WRC receives any part of a response, it verifies the class this response belongs to, and invokes the ACC adjuster to update that class's ACC counter. Once the received data includes the last packet of a response, W would be decreased by one to imply a request is fully answered. If now is the end of a *recomputing interval*, the utilization U is re-computed. The next step after adjusting W_{max} is invoking the *request scheduler* to release requests as more as

possible, till W_{max} requests have been sent or no more request can be send.

```
WRC
rsp = Rcv()
Qno = VerifyQ(rsp)
len = Size(rsp)
AccAdj(TypeRsp, Qno, len)
if ( LastPkt(rsp) ) {
     W --
     rcvd_rsp++
     if ( rcvd_rsp == FxWmax ) {
        ComputeUtil()
        AdjustMaxWin()
     }
    while ( (W < Wmax) && (backlogged) )
        Scheduler()
}
```

Fig. 3 Procedure of window-based service-rate controller

3.4 Examples

This section gives two examples displaying the scheduling behavior in FQRS.

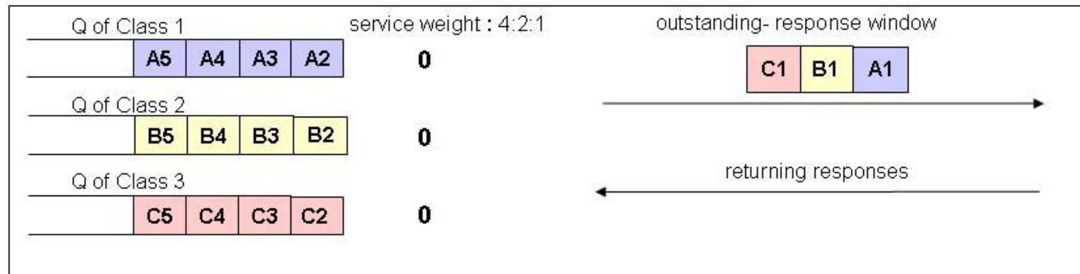
3.4.1 Backlogged Traffic

Traffic of a class is backlogged at time *t* if some requests are queued at this class's queue. Fig. 4 shows an example of backlogged traffic. Consider the case that three classes compete for the bandwidth of the bottlenecked link. The ratio of their service weights is 4:2:1.

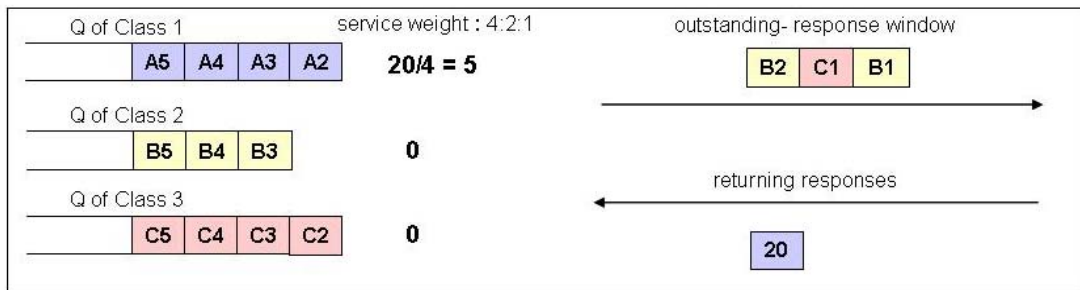
As shown in Fig. 4(a), at time 0, the values of all three ACC counters are 0. Some requests have already arrived. Also, three requests, A1, B1, and C1, have been sent out from Q_A, Q_B, and Q_C, respectively. Suppose the size of the window is three (W_{max} = 3). Hence, all slots of the window are occupied by the three outstanding responses A1, B1, and C1.

As shown in Fig. 4(b), when the first 20-byte response corresponding to the request A1 returns, the ACC counter of class A (ACC_A) is calculated as 20/4, and a slot of the window is freed. After that, the scheduler selects next class queue with

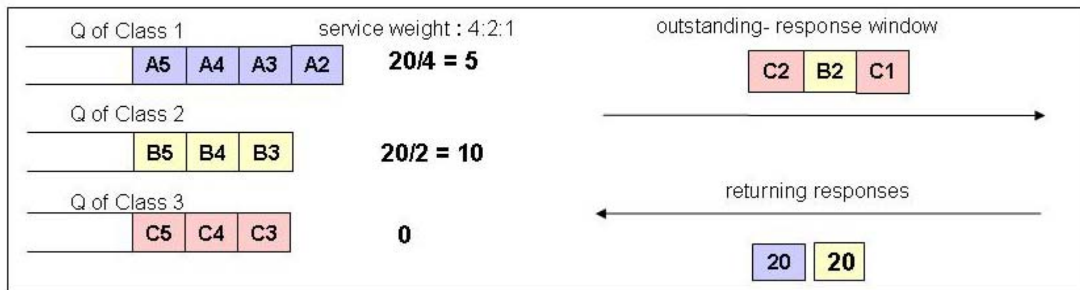
minimum ACC counter to serve, i.e. Q_B. The HOL request B2 of Q_B occupies the freed slot of the window and is sent out now. Next, as shown in Fig. 4(c), the response corresponding to the request B1 returns. A slot of the window is freed again and ACC_B is updated as 20/2. The HOL request of Q_C, C2, is scheduled.



(a). Initial Status



(b). The snapshot when response A1 returns



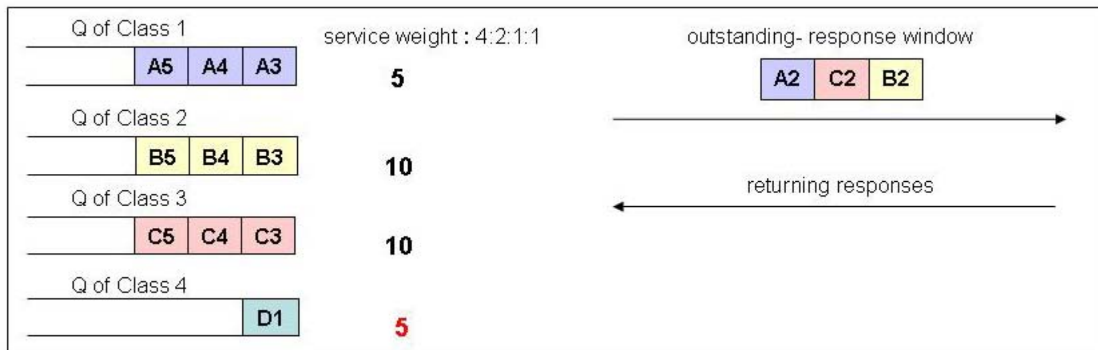
(c). The snapshot when response B1 returns

Fig. 4 Example for FQRS in handling backlogged traffic.

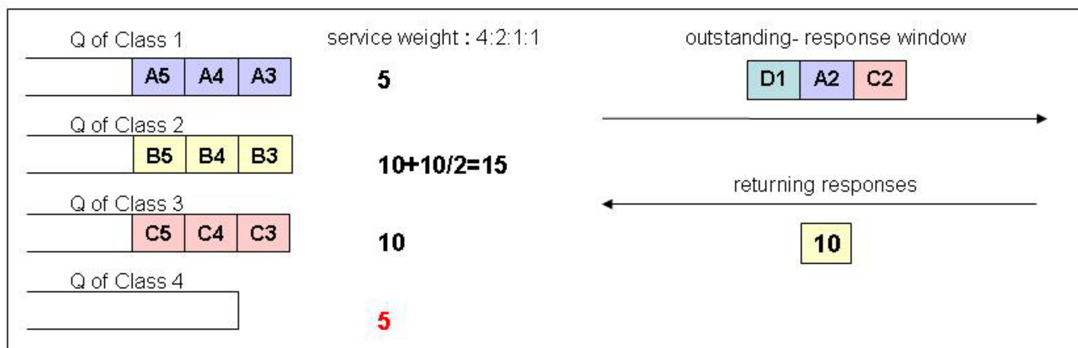
3.4.2 Traffic from a Reviving Class

Traffic of a class is reviving at time *t* meanwhile some requests arrive to an empty class queue. Fig. 5 illustrates an example of reviving traffic. Assume 3 class queues, Q_A, Q_B, and Q_C, have backlogged traffic, and the Q_D is empty for a while. The ratio of their service weights is 4:2:1:1.

As shown in Fig. 5(a), at time *t*, the values of the four ACC counters are 5, 10, 10, and 0, respectively. Suppose the request D1 belonging to Q_D arrives at the time a little later than time *t*. After the D1 enters the Q_D, ACC_D is set as 5, the minimum of all other ACC counters. Then, as shown in Fig. 5(b), once a response, for example, the 10-byte response corresponding to the request B2 returns, Q_D can be served instantly. That is, D1 is the next request to be sent out as soon as a slot of the window is free.



(a). D1 arrives in empty queue Q_4



(b). The snapshot when response B2 returns

Fig. 5 Example for FQRS in handling requests which meet an empty class queue. The ACC_D is reset as the minimum among all other ACC counters

3.5 Discussions for Exceptive Cases

The FQRS is designed under the assumption that all uplink traffic is requests and downlink traffic is responses. Also, all responses are triggered by the FQRS-released requests. However, the exceptive traffic does coexist with the assumed traffic on the real network environment. The exceptive traffic is classified into three types for conveniently discussion.

3.5.1 Downlink exceptive traffic belonging to some classes

The first type of traffic means the exceptive traffic that can be classified into the pre-defined user classes. For example, POP3 mail traffic for someone host in $Class_i$. The traffic may still be the downlink responses, but their requests are not recognized by the implementation of FQRS. It is possible since only the web request is recognizable for the present implementation of FQRS.

FQRS regards these exceptive packets as the received service of classes. When the packets not triggered by an (recognized) request arrive from the Internet, their sizes are accumulated into the AC of the class which the packets belong to, as other response packets triggered by requests. For example, if a $Class_i$ host receives a crowd of such packets from the Internet, the sizes of packets would be accumulated into AC_i. The additional increased value in AC_i because of these unexpected packets brings the less requests can be sent out from the $Class_i$ by FQRS.

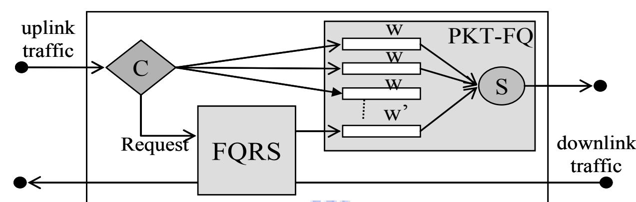
3.5.2 Downlink exceptive traffic not belonging to any class

Exceptive traffic not belonging to any class may include requests from outside network for the responses provided by the intranet servers, or the malicious attacks. The former case is possible for the enterprises having web servers for their customers. The exceptive traffic contributed from such request is small usually, compared with other responses traffic running on the downlink. The latter case may rudely and fully fill the downlink, resulting in the failure for all transmissions. However, the latter case is a security problem and out of the scope of this work.

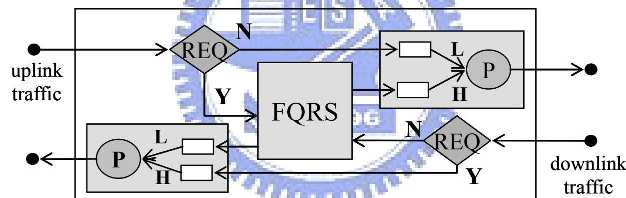
This exceptive traffic is considered by the WRC in FQRS as other assumed response traffic, when WRC monitors the utilization of the downlink to adjust W_{max}. More ratio of exceptive traffic may only bring a smaller W_{max}. That is, WRC may think such a small W_{max} is enough to fully utilize the downlink bandwidth. Fairness between classes is not affected in this case.

3.5.3 Uplink exceptive traffic

The traffic may include uplink responses and traffic actively sent from the internal hosts. If this traffic turns the uplink to a bottleneck, a traditional FQ discipline at the access gateway is suggested first. FQRS and the uplink FQ discipline could be coexist well, as shown in Fig. 6(a). Also, if the fairness of uplink is not a concern, FQRS + Priority Queues would be a simple solution, as shown in Fig. 6(b). The solution gives the request traffic high priority since they are smaller than responses usually.



(a) The architecture for managing the bottlenecked uplink



(b) The architecture for network without uplink QoS

Fig. 6 Two potential integrated architectures for handling the network when exceptive traffic coexists

Chapter 4 Simulation Results

This chapter verifies the effects of the FQRS through simulation by *ns-2* [18] in terms of the proportional fairness and sharing, user-perceived latency, the relationship between the access link utilization and value of W_{max}, and the choice of F and U_{max}.

4.1 Topology

The HTTP/Cache class in *ns-2* acts as a web proxy cache, and sits between clients and web servers. It intercepts the requests sent from clients and forwards them to the remote servers if the requested data is not cached yet. Therefore, this work disables the cache function in HTTP/Cache and implements FQRS in the class.

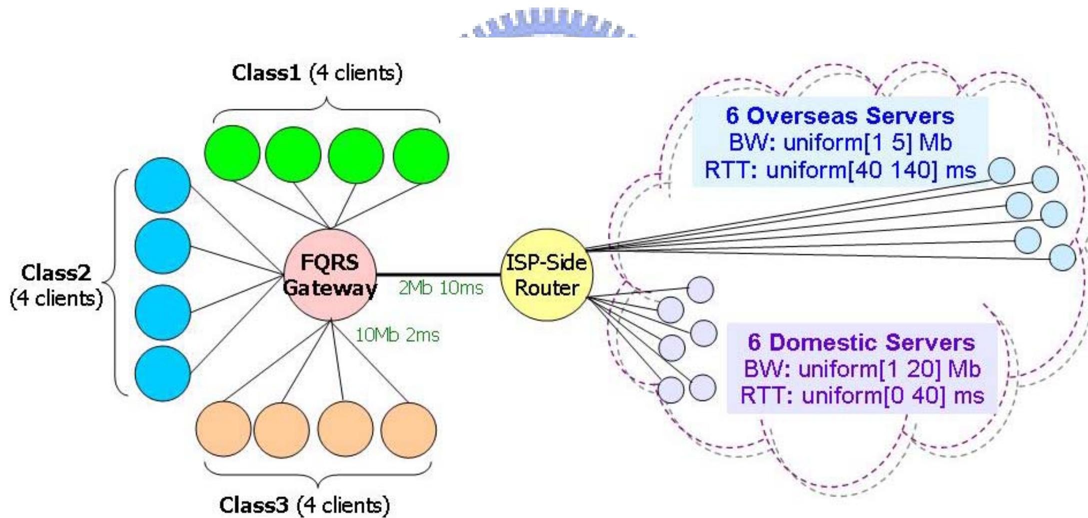


Fig. 7 Simulation topology for three classes with service ratio 4:2:1

Figure 7 shows the topology used in the simulation. The FQRS gateway provides three classes, Class1, Class2, and Class3, with the service ratio 4:2:1. Each class involves three clients and all clients connect to remote web servers through the FQRS gateway. The link between the FQRS gateway and every client is 10Mbps with 2 ms propagation delay. The access link connecting the FQRS gateway and the ISP router is configured as 2Mbps capacity with 10ms propagation delay. The ISP router connects

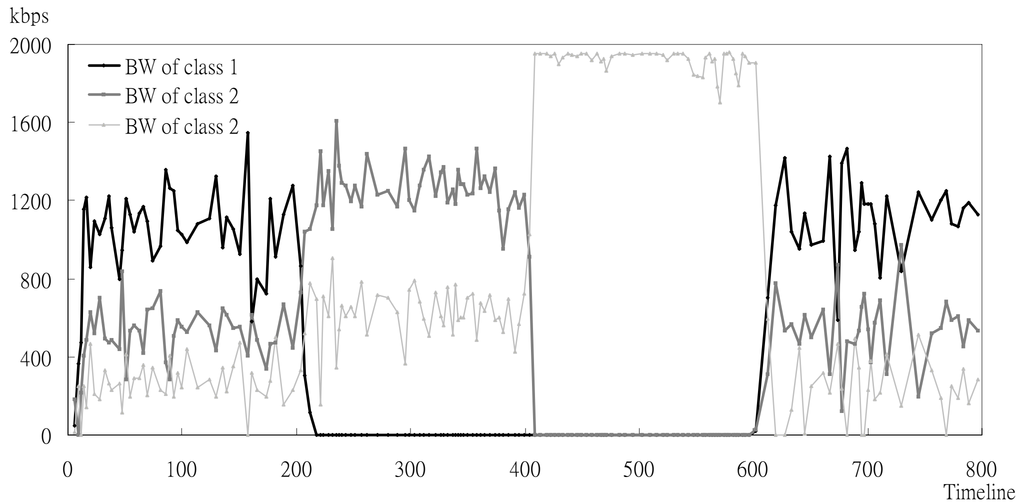
to twelve servers with twelve independent links. These servers are grouped into two server farms, each involving six servers. The two farms represent overseas servers and domestic servers. Links to these servers have a *uniform* distribution, as shown in Fig. 7. By the statistics from the real Internet [19], the web page size is a lognormal-distributed random variable X with ln(X)=9.357 and σ =1.318. The average response size is 27,656 bytes.

4.2 Weighted Fairness and Sharing

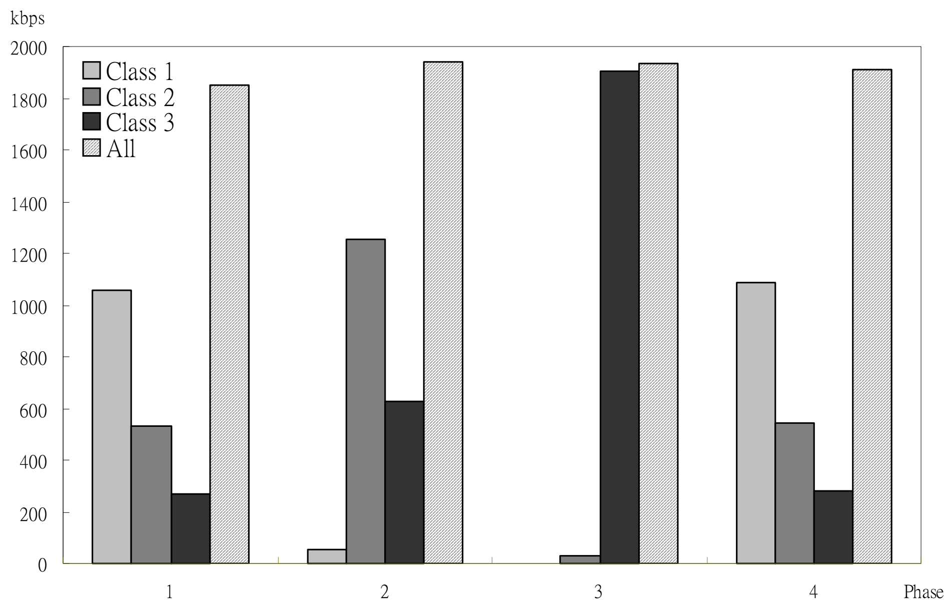
We demonstrate that FQRS provides proportional bandwidth between classes and the idle bandwidth can be shared by active classes. Four phases are included in the simulation and the duration of each phase is 200 seconds. In the first phase, all of the three classes have backlogged requests. In the next two phases, Class1 and Class2 stops requesting separately, and then both of them have backlogged requests again in the last phase.

Fig. 8(a) depicts the throughput of each class in every *recomputing interval* over the 800 seconds. Fig. 8(b) shows the average throughput in each phase. During the first phase, the three classes get proportional bandwidth in the service ratio 4:2:1. We measure the short-term fairness of service ratio among classes through fairness index [22]. The fairness index, defined as $(\sum_{i=1}^{K} x_i)^2 \ / \ (K\sum_{i=1}^{K} x_i^{\ 2})$ where $\sum_{i=1}^{K} x_i$ represents the sending rates of competing flows and K is the number of sampling, in this phase is 0.89. In the second phase, the idle bandwidth freed by Class1 is shared by Class2 and Class3 proportionally. Both of the bandwidth obtained by Class2 and Class3 increase in this phase, and the usage ratio between them is still 2:1. After Class2 stopping requesting in the third phase, Class3 occupies all bandwidth until the end of this phase. During the second and the third phase Class 1 and Class 2 still obtain a bit of bandwidth separately due to their unfinished transactions. Once all idle classes have

requests again in the last phase, the three classes obtain the bandwidth in the expected proportion, 4:2:1, again. Compare the total bandwidth usage in the first phase and the fourth phase, and the one in the first phase is lower because it costs time to raise the bandwidth utilization in the initial state.



(a) Bandwidth usage of three classes depicted every account interval



(b) Average bandwidth usage of three classes in four phases

Fig. 8 bandwidth usage of three classes with service ratio 4:2:1 in four phases

4.3 Lower Average Latency

When a FQRS gateway is deployed, the user-perceived latency can be

decomposed into transmission time and queuing time. The transmission time is defined as the time spent on transmitting both of requests and responses between the FQRS gateway and remote servers. The queuing time represents the time when the request is queued in the FQRS gateway. The time in transmitting packets between clients and the FQRS gateway is small and can be ignored. Besides, server answering time is always zero in *ns-2*.

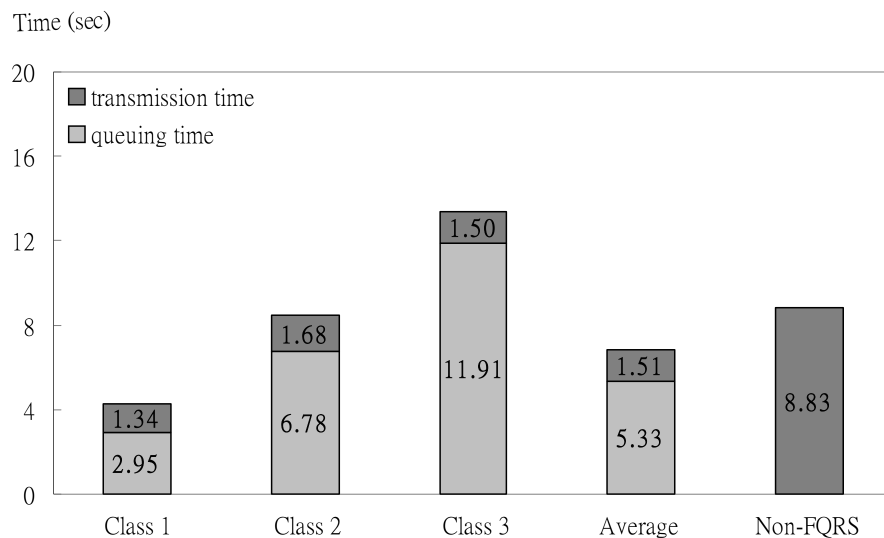


Fig. 9 User-perceived latency comparison by decomposing time factors: queuing time and transmission time

The simulation scenario here is the same as that used in the first phase in section 4.1. Fig. 9 illustrates the decomposition of user-perceived latency for the three classes, the average latency among all classes, and the latency if no FQRS is deployed, denoted as non-FQRS. First, by comparing the left three bars, the different user-perceived latencies are experienced by the three classes. They have different queuing time in FQRS since they have different weights.

Second, by comparing the right two bars, the average latency (6.76 secs) in FQRS is shorter than that in non-FQRS (8.83 secs) by 23.44%. It is because the average transmission time in FQRS (1.5 secs) is far shorter than that in non-FQRS (8.83 secs). The transmission time in FQRS is reduced as a result of reasonable

number of concurrent outstanding responses. Section 4.5 with Fig. 11 would further support the hypothesis.

4.4 Adjustment of Window Size

This section observes the relationship between access link utilization and the value of W_{max}. Clients in the three classes send requests in the 600-second duration except the middle 200 seconds. During the middle 200 seconds, clients in Class1 and Class2 stop sending requests. That comes out insufficient requests so that the FQRS gateway may have no request to send out.

Fig. 10 reveals the relation between the access link utilization and the value of W_{max}. The utilization of access link stays around 0.95 as the expected U_{max} in the first and last 200-second periods because of sufficient arrival requests.

In the 200th~400th sec, the utilization falls apparently due to the insufficient arrival traffic, and the value of W_{max} is constant. It is in vain to raise the value of W_{max} when the incoming requests are too few to occupy all window slots. Therefore the value of W_{max} keeps steady as described in equation 4. Because of insufficient arrival requests, there are often some window slots are free. Once the last packet of a response returns, queued requests can be scheduled out as more as possible until no more requests can be sent or no more window slot is available. That is why the value of W in Fig. 10 varies with a wide range during the 200th~400th sec.

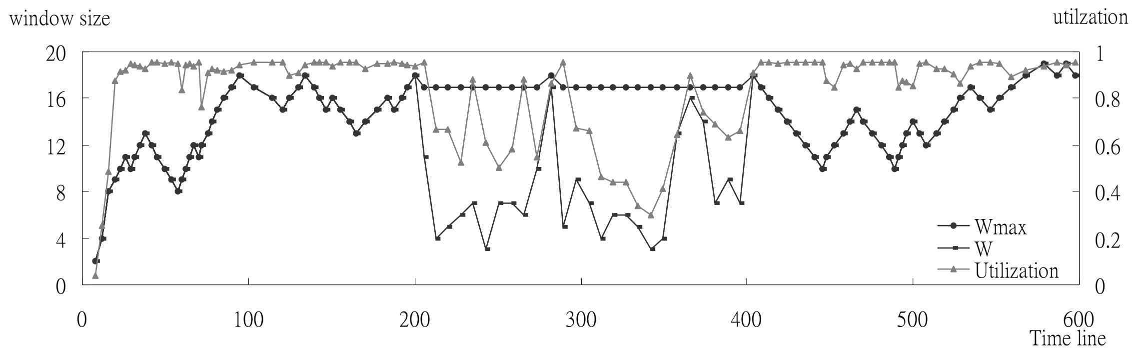


Fig. 10 The size of W_{max} is steady in case of insufficient traffic.

4.5 Choice of F and U_{max}

This section examines the choice of F and U_{max}, two parameters can be configured by administrators. This examination starts with the effect of F, and then on the effect of U_{max}.

1) Effect of F: According to the examination on the effect of F between 1 and 16 when U_{max} is assigned to 0.8, 0.9, 0.95, and 0.99, the bandwidth utilization is not susceptible to the value of F. When the value of F is smaller than 16, the degree of bandwidth utilization can reach 95% of expected U_{max}, even though U_{max} is high as 99%. From this examination, it is suggested that value of F is set between 4 and 8 because they can come out best bandwidth utilization with respect to the expected U_{max}.

2) Effect of U_{max}: Fig. 11 depicts the user-perceived latency, the queuing time spent in FQRS, and the number of queued packet at the ISP-side router when U_{max} is assigned to 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, and 0.99 while F is assigned to 4. As U_{max} increases, the user-perceived latency and the queuing time reduce meanwhile the number of packets queued in ISP router rises. Raising U_{max} follows shorter user-perceived latency, because more responses can be transmitted concurrent and the bandwidth can be utilized more. However, the raise also causes packets to be queued in ISP router because of less free bandwidth for eliminating the queued packets as U_{max} is high. From observation of Fig. 11, the value of U_{max} is suggested to be set between 90% and 95%.

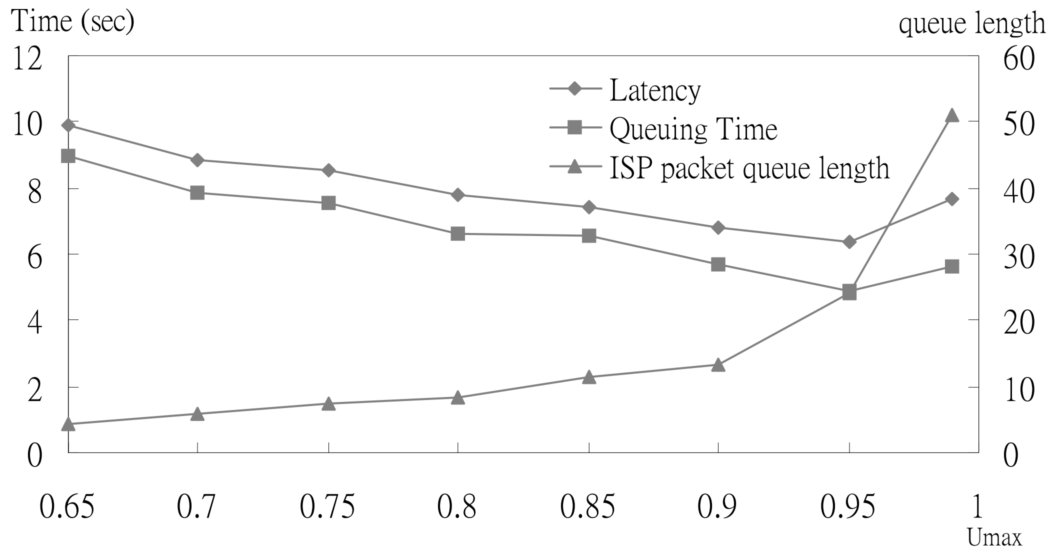


Fig. 11 Variations of user-perceived latency, queuing time, and the number of packets queued in ISP-side router under various U_{max}

Chapter 5 Implementation

We implement FQRS in Squid [16], which is an open source package of web proxy cache, and perform field trial in an open network environment.

5.1 Architecture

Fig. 12 illustrates the transaction flow in Squid, and Table 1 lists the functions used in the transaction flow. Data structures are displayed in dotted circles. The data structure *ConnStateData* maintains the status of a connection. It is built when *httpAccept()* accepts a new connection from a client. The two data structures *clientHttpRequest* and *request_t* keep a request's status. They are established when *clientReqdRequest()* parses a request. After verifying access right and checking cache, Squid starts to forward this request to remote server by the procedure *clientProcessRequest()* according the request's HTTP method.

The three components of FQRS, *request receiver* (*FQRS_ReqRcv()*) , *request scheduler* (*FQRS_Scheduler()*) and *ACC adjuster* (*FQRS_AccAdj()*), are shaped in double rectangles in Fig. 12. They are implemented after cache checking and before forwarding. For each class, FQRS in Squid uses a link list to maintain its arrival requests, as shown in the dotted circle A.

When the response returns in *httpReadReply()*, the FQRS component *WRC* (*FQRS_WRC()*) updates the value of the corresponding ACC counter, and refreshes the maximum allowed concurrent outstanding responses if needed. Once the responses finishes, *fwdComplete()* forwards the response to the client who requested it.

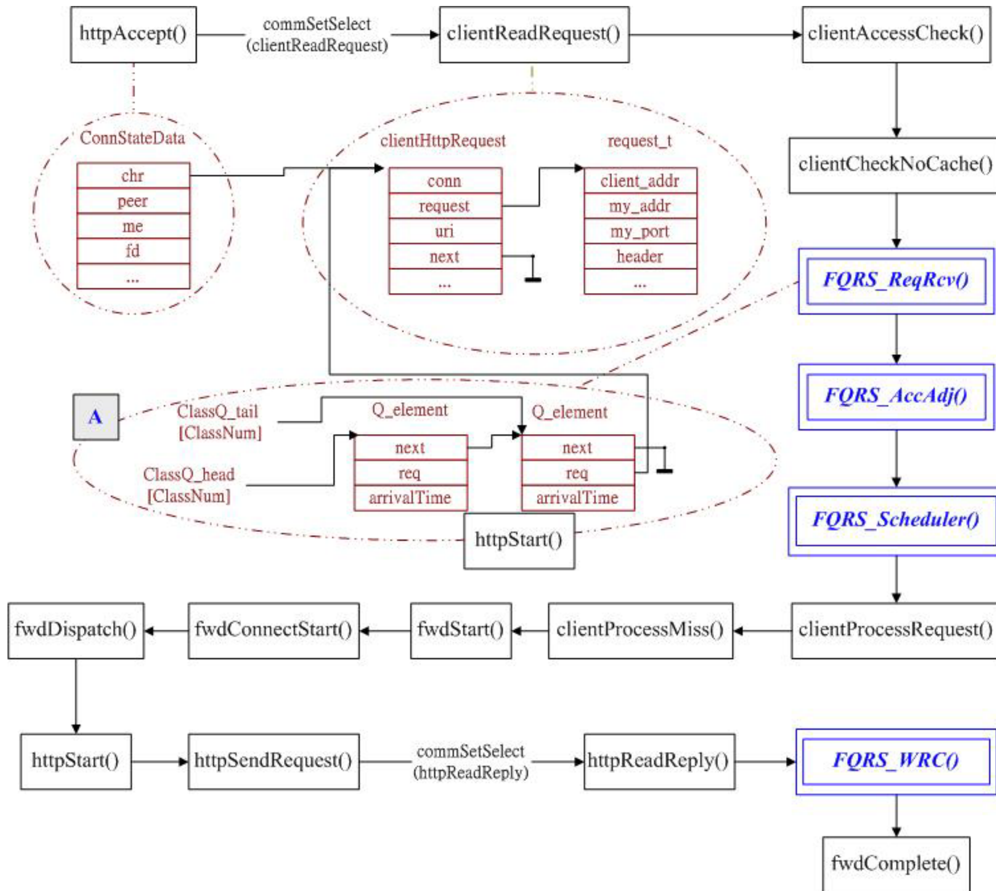


Fig. 12 The transaction flow in Squid. The components of FQRS are shape in double rectangles. Dotted rings circle the data structures

Table 1 The Functions of transaction flow functions in Squid

Function Name	Function Description
*httpAccept (sock, void *)*	Accept a new connection on HTTP socket
clientReadRequest (fd, ConnStateData)*	Parse received HTTP request
*clientAccessCheck (clientHttpRequest *)*	Verify client's access right
*clientCheckNoCache (clientHttpRequest *)*	Check if forced to never cache
*FQRS_ReqRcv(srv_ip, clientHttpRequest *)*	Classify request, adjust ACC value
FQRS_AccAdj()	Adjust ACC counters
FQRS_Scheduler()	Schedule a request out
*clientProcessRequest(clientHttpRequest *)*	Verify connection method
*clientProcessMiss(clientHttpRequest *)*	Prepare to fetch the object due to cache miss
*fwdStart(fd, request_t *)*	Start to forward the request to remote server
fwdConnectStart(fwdState)	Establish connection to remote server
fwdDispatch(fwdState)	Dispatch according to protocol type
httpStart(fwdState)	Start HTTP action
httpSendRequest(httpState)	Write request to remote server

httpReadReply()	Read reply
FQRS_WRC()	Controlling the sending rate of requests
fwdComplete()	Forward the response to the whom asks it

5.2 Field Trial

Fig. 13 illustrates the test bed for evaluating the FQRS in Squid. Avalanche [20] is an application-layer traffic generator. It emulates the behaviors of multiple clients and sends requests to the web server in the Internet. Avalanche imports a URL list, which is a historical URL record logged by an enterprise in a couple of days.

The access gateway installed with FQRS is configured as a transparent proxy with iptables [21]. All HTTP requests destined to the port 80 are directed to the port 3128, the service port of Squid. A layer 3 switch is acts as the ISP-side router. The bandwidth of the access link between the access gateway and the layer 3 switch is limited to 2Mbps. As the configuration in simulation, three classes are provided with service ratio 4:2:1. Notably, the cache function is disabled to avoid responses are got from the local cache or remote caches. The effects of FQRS Squid are observed in terms of weighted fairness, user-perceived latency, and CPU loading as follows.

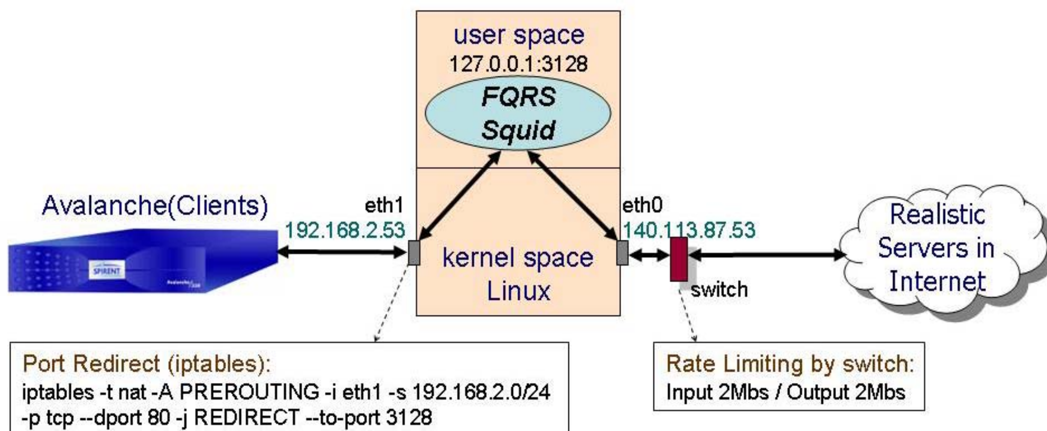


Fig. 13 Test bed for field trial in Squid

1) Weighted Fairness: The amounts of bandwidth allocated to three classes for a 200-second test round are 1.03 Mbps, 0.52 Mbps, and 0.26 Mbps, respectively, when backlogged requests are applied. It quite obeys the configured service ratio 4:2:1. The

fairness index [22] measured here is 0.87.

2) User-Perceived Latency: Table 2 shows the per-request latency provided by the original Squid, the FQRS Squid, and the no-Squid case. The no-Squid case represents all packets are simply forwarded. First, we compare the former two cases. The FQRS Squid reduces (1686-1174)/1686, or 30%, of the average user-perceived latency in the original Squid case, although the user-perceived latency in FQRS includes the additional queuing time, 515.5 ms. Next, by comparing the latter two cases, we see the user-perceived latency in FQRS is much longer than the one in the no-Squid case. It is believed the longer latency is resulted from the overloads of user-space processing and TCP-connection interception. Appling the FQRS framework in kernel space without additional overload is the future work.

Table 2 User-perceived latency comparisons. FQRS Squid spends additional queuing time.

Items	Original Squid	FQRS Squid	No Squid
User-Perceived Latency (ms/request)	1686.1	1174.9 (includes queuing time 515.5)	473.7

3) CPU Loading: Table 3 shows the benchmark results on percentage of CPU usage percentage in the FQRS Squid process and the original Squid process with equal throughput. As expected, the CPU loading increases as the number of classes or the access link bandwidth increases. Notably, the loading under the FQRS is always lower than that under the original Squid. Under the original Squid, all requests are released by the proxy immediately, bringing many concurrent transactions. However, the appropriate number of concurrent transactions is allowed by FQRS. It is believed that the number of concurrent transactions dominates the cost of CPU computing under the same throughput.

Table 3 Comparison between FQRS and the original Squid in percentage of CPU time

	Throughput	
	2 Mbps	10 Mbps
FQRS Squid with 10 Classes	22.4	28.17
FQRS Squid with 100 Classes	23.01	29.02
Original Squid	31.61	42.45

Chapter 6 Conclusions and Future Works

This work reveals two problems occurring in scheduling requests with fair queuing disciplines. First, the bandwidth of bottlenecked link is resumed by responses instead of the scheduled requests. Besides, the bandwidth of bottlenecked link may be exploited by multiple responses simultaneously. The releasing rate of requests has to be designed well. Second, response size is varied and unavailable until the first packet of the response returns while transmission rate of a response varies over time. Therefore it is not workable to decide the releasing order by service completion time.

Based on the discussion of the two problems, a fair-queuing based request scheduling (FQRS) discipline is proposed to manage the access link bandwidth at user-side access gateway. To achieve high bandwidth utilization, FQRS adopts window-based service-rate control (WRC) to determine the releasing rate of requests and the number of concurrent outstanding responses. To perform proportional fairness and bandwidth sharing, Request-based fair queuing (RFQ) in FQRS serves the class queue according to the accumulated received normalized service amount in bytes of each class.

The results in the simulation and the field trial of FQRS show the bandwidth usage between classes conforms to the targeted ratio and the idle bandwidth is proportionally shared by all active classes. The fairness index, which presents the short-term fairness among classes, is 0.89 and 0.87 respectively. Besides, the FQRS reduces 23.44% and 30% of user-perceived latency individually because the number of concurrent transmissions is controlled, even this control may queue the requests in the FQRS gateway. The examination on F of *recomputing interval* suggests a value from 4 to 8 can reach best bandwidth utilization with respect to the expected

utilization U_{max}, and the examination on U_{max} suggests a value from 90% to 95% can come out shortest user-perceived latency.

Our future plans will start from formally proving on service fairness and further investigating the abnormal case as described in section 3.5.3. After that, we will verify the outcomes of FQRS when multiple applications coexist and when traffic not triggered by inside requests is mixed. Besides, to reduce the overhead, implementing FQRS on kernel space is required.

References

[1] A. K. Parekh, R. G. Gallager, "A generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node case," IEEE/ACM transactions on networking, pages 344-357, June 1993.

[2] J. Golestani, "A Self-Clocked Fair Queueing Scheme for Broadband Applications," In proceedings of the IEEE INFOCOM, Toronto, June 1994.

[3] P. Goyal, H. Vin, and H. Chen, "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," Proceedings of the ACM SIGCOMM, August 1996.

[4] M. Conti, M. Kumar, S. K. Das, and B. A. Shirazi, "Quality of Service Issues in Internet Web Services," IEEE Transactions on Computers, vol.51, no.6, June 2002.

[5] R. Pandey, J. Fritz Barnes, and R. Fritz Barnes, "Supporting Quality of Service in HTTP Servers," Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, pp. 247-256, 1998.

[6] N. Bhatti, A. Bouch, A. Kuchinsky, "Integrating User-Perceived Quality into Web Server Design," Proceedings of the 9th International World Wide Web Conference, 2000.

[7] L. Cherkasova and P. Phaa, "Session Based Admission Control: a Mechanism for Web QoS," Proceedings of the International Workshop on Quality of Service, 1999.

[8] L. Eggert and J. Heidemann, "Application-Level Differentiated Services for Web Servers," World Wide Web Journal, vol. 3, issue 2, pp. 133-142, 1999.

[9] T. P. Kelly, S. Jamin, and J. MacKie-Mason, "Variable QoS from Shared Web Caches: User-Centered Design and Value-Sensitive Replacement," Proceedings of IEEE Conference on Computer Communications, 2002.

[10] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao, "An Adaptive Control Framework for QoS Guarantees and its Application to Differentiated Caching Services," Proceedings of the International Workshop on Quality of Service, 2002.

[11] V. Cardellini, E. Casalicchio, M. Colajanni, and M. Mambelli, "Enhancing a Web-Server Cluster with Quality of Service Mechanisms," Proceedings of IEEE International Performance Computing and Communications Conference, 2002.

[12] E. Casalicchio and M. Colajanni, "A Client-Aware Dispatching Algorithm for Web Clusters Providing Multiple Services," Proceedings of the 10th International World Wide Web Conference, 2001.

[13] C. Li, G. Peng, K. Gopalan, T. Chiuch, "Performance Guarantee for Cluster-Based Internet Services," State University of Stony Brook, May 2001.
[14] Y. H. Lin, Y. D. Lin, "Request Scheduling for Web QoS at Edge Devices," NCTU, June 2003.
[15] M. K. Ouyang, Y. D. Lin, "Request Scheduling for Differentiated QoS at Access Gateway," NCTU, June 2004.
[16] Squid Web Proxy Cache, http://www.squid-cache.org/.
[17] M. Shreedhar, and G. Varghese, "Efficient Fair Queuing Using Deficit Round-Robin," IEEE/ACM transactions on networking vol. 4, no. 3, June 1996.
[18] The Network Simulator - ns-2, http://www.isi.edu/nsnam/ns/.
[19] 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.
[20] Avalanche, http://www.spirentcom.com/analysis/product_line.cfm?PL=32.
[21] The netfilter/iptables project, http://www.netfilter.org/.
[22] R. Jain, K. K. Ramarkrishnan, and D. M. Chiu, "Congestion avoidance in computer networks with a connectionless network layer," Tech. Rep. DEC-TR-506, DEC, Aug, 1987.
[23] "DWSR: A New One-Way L7 Web Switch Architecture," NCTU, 2004.
[24] J. M. Blanquer, B. Özden, "Fair Queuing for Aggregated Multiple Links," ACM SIGCOMM, August 2001.