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

多重資源請求排程

Multiple-resource Request Scheduling
for Differentiated QoS at Website Gateway

研 究 生：馮若華

指導教授：林盈達 教授

中 華 民 國 九 十 四 年 六 月

在網站閘道器上提供差別服務品質之多重資源請求排程

學生：馮若華　　指導教授：林盈達

國立交通大學資訊科學研究所

摘 要

差別服務品質是網站經營者提供給客戶不同層級服務的一種方式，傳統 HTTP 請求排程的方法可以達成這個目標，但是它們排程請求只能管理一種伺服器資源，例如 CPU 或 Disk I/O，但實際上伺服器在處理一個請求時會消耗多種資源，單一資源的排程法會導致資源的浪費或系統的過載。本論文提出了一個名為 mQoS 的多重資源請求排程演算法來將伺服器上的多種資源使用量作差別管理，mQoS 排程演算法是部署在網路閘道器上，因此它對客戶端及伺服器端是透明的。mQoS 排程器中是由數個子排程器及一個主排程器所組成，每一個子排程器管理一種伺服器資源以便在各個服務類別之間差別資源使用量，主排程器則檢查每種伺服器資源的可使用量並觸發適當的子排程器以平衡各伺服器資源的使用。mQoS 排程演算法的設計想法是源自於傳統的 Deficit Round Robin 封包排程演算法，每個服務類別都對應到一個 deficit counter 以紀錄其未使用的伺服器資源量，而子排程器中的 deficit counter 可以被任何一個子排程器減值，主要原因為請求是消耗多重資源而非單一資源。mQoS 閘道器的實作是架於 Squid 及 Linux 之上。在性能評估方面，將比較 mQoS、無排程(nQoS)及單一資源排程(sQoS)以展現 mQoS 排程的效果。在以差別服務比例 6:3:1 分配給三個服務類別的例子中，mQoS 排程準確地分配各種伺服器資源，另外，mQoS 排程的總伺服器吞吐量比 sQoS 排程增進了 21%。mQoS 的平均使用者察覺延遲比其他兩種方式較短。

關鍵字：多重資源、差別服務、請求排程

Multiple-resource Request Scheduling for Differentiated QoS at Website Gateway

Student: Ruo-Hua Feng Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science
National Chiao Tung University

Abstract

Differentiated quality of service is a way for a Website operator to provide different service levels to its clients. Traditional HTTP request scheduling schemes can achieve this, but they schedule requests to manage only one server resource, such as CPU or Disk I/O. Actually, processing a request on the server will consume multiple resources. In this paper, a multiple-resource request scheduling algorithm, called mQoS, for differentiating the utilization of the server resources is presented. The mQoS scheduler consists of several sub-schedulers and a main scheduler. Each sub-scheduler manages a server resource to differentiate the utilization among the classes. The main scheduler checks the availability of every server resource and triggers an appropriate sub-scheduler to balance the utilization of server resources. The idea of the mQoS scheduling algorithm is derived from the traditional deficit round-robin method. There are some deficit counters in a sub-scheduler. However, a deficit counter of a sub-scheduler can be decremented by other sub-schedulers because a request would consume multiple resources. The implementation of the mQoS gateway is based on the Squid and Linux. In the evaluation, the mQoS scheduling is compared with no scheduling (nQoS) and single-resource request scheduling (sQoS). The mQoS scheduling reveals the accurate differentiation on every server resource. In addition, the total server throughput in the mQoS scheme is improved by 21%, compared with the sQoS s. The average user-perceived latency of the mQoS scheduling is shorter than other schemes.

Keywords: Multiple Resources, Request Scheduling, Service Differentiation

Contents

Chapter 1. Introduction **1**

Chapter 2. Problems of Server Resource Management **4**

Chapter 3. The mQoS Gateway Architecture and Scheduling Algorithm **8**

3.1 Server Prober 9

3.2 Content-aware Request Classifier 11

3.3 Multiple-resource Request Scheduler 11

3.4 Multiple-Resource Request Scheduling Algorithm 13

Chapter 4. Implementation and Evaluation **21**

4.1 Implementation 21

4.2 Evaluation 22

4.3 Differentiation on the Resource Utilization 23

4.4 Differentiation on the Server Throughput 25

4.5 Resource Sharing 27

4.6 Differentiation on the User-perceived Latency 27

Chapter 5. Conclusion and Future Work **30**

References **32**

List of Figures

Figure 1. Server resource utilization under different scheduling schemes. 7

Figure 2. Architecture of the mQoS gateway. 9

Figure 3. mQoS scheduler. 13

Figure 4. Flowchart of the mQoS scheduling algorithm. 15

Figure 5. Pseudo Code of the mQoS scheduling algorithm. 18

Figure 6. An example of the mQoS scheduling. 20

Figure 7. Implementation of the mQoS gateway on the Squid. 22

Figure 8. Evaluation environment 23

Figure 9. Server resource utilization of the nQoS, sQoS, and mQoS scheduling. 24

Figure 10. Server throughputs of the nQoS, sQoS, and mQoS scheduling. 26

Figure 11. Types of outstanding requests by the sQoS and mQoS scheduling. 26

Figure 12. Resource sharing of the nQoS, sQoS, and mQoS scheduling. 27

Figure 13. User-perceived latency of the nQoS, sQoS, and mQoS scheduling. 28

Figure 14. Decomposition of the user-perceived latency in the mQoS scheduling. 29

Chapter 1. Introduction

Web Quality of Service (QoS) is a way for a Web service provider to differentiate its service levels to users. Through *service differentiation*, a Web service provider can allow a specific group of users, e.g. paid users, to get better server throughput or user-perceived latency than other general users. There are many ways of enforcing Web QoS. The effort of some past researches was to modify the system kernel or the server daemon of a Web server, a caching proxy, or a cluster dispatcher for service differentiation. These QoS-enabled boxes intercept HTTP requests, perform request classification and schedule requests for dealing with the bottlenecked resource, such as throughput or CPU utilization.

There are two issues in the above schemes. The first issue is where to deploy a QoS-enabled box. Many researches have been proposed in modifying the system kernel [1] or server daemon [2][3] of a Web server to have the capability of scheduling HTTP requests. However, this solution is hard to be deployed on a non-open operating system or server daemon. Some researches have been proposed in enforcing request scheduling on a dispatcher of a cluster server [4-6]. The QoS-enabled dispatcher schedules requests to the backend servers in a weighted round-robin fashion or according to the server loads. Some researches have proposed QoS-enabled content adaptation [7][8] or cache replacement algorithms [9] on caching proxies instead of request scheduling for service differentiation.

The second issue is what resource for a request scheduling to manage for. Common request scheduling schemes schedule requests by managing the bottlenecked resource, such as the number of requests per second. This request scheduler seems to perform the single-resource scheduling, which has a blind spot.

Processing a request on a server needs to consume multiple resources, e.g. CPU, disk I/O, and bandwidth, rather than a single resource. In the single-resource scheduling, some resources may be wasted, when the managed resource is well utilized. A request scheduler should well utilize all resources by scheduling requests for managing all resource utilization. Some researches have discussed *multiple-resource request scheduling*, but many of them are applied on grid computing and multimedia applications [10-12], few on HTTP request scheduling [13-15].

Considering the issues of QoS deployment and multiple-resource request scheduling, this paper presents a multiple-resource request scheduling algorithm called mQoS, which is deployed at a *Website gateway* for controlling the requests toward a Web server. Today's gateways can perform firewall packet inspection, intrusion detection, virus scanning, and so on. A Websites operator can deploy a gateway for preventing attacks and providing value-added services. Hence, enforcing request scheduling at a Website gateway is practical, and it can provide service differentiation without any modification on clients and the server.

There are three main functions in the mQoS gateway: *request profiling and server profiling*, *content-aware request classification*, and *mQoS scheduling*. The request profiling finds out the amounts of the server resources consumed by a request, whereas the server profiling measures the capacities of the server resources. The request classification mechanism inspects the headers or payloads of requests and puts requests into proper class queues. Specially, a service class has several queues, each of which stores specific resource-intensive requests. That is, it will be m*n queues, when there are m service classes and n server resources. The mQoS scheduling, derived from the *Deficit Round Robin* (DRR) scheduling [16], composed of one main scheduler and several sub-schedulers. One sub-scheduler, which has some deficit counters, manages one server resource. However, differing from the traditional DRR

scheduling, a deficit counter of a class in a sub-scheduler can be decremented by any sub-scheduler because a request would consume multiple resources rather than a single resource. In addition, the main scheduler maintains the availability of the server resources in the resource availability counters. The main scheduler hence can know which resource is the most available and then triggers the corresponding sub-scheduler to service specific resource-intensive requests.

The mQoS gateway is implemented on Squid and Linux. The request and response modules of Squid are modified to be capable of classifying and scheduling requests. In the evaluation, the mQoS scheduling is compared with no scheduling (nQoS) and single-resource request scheduling (sQoS). The resource utilization, server throughput, and user-perceived latency of every scheduling are measured to demonstrate the effect of the mQoS scheduling. From the test results, the mQoS scheduling reveals its capabilities of differentiating server resource utilization, maximizing the total server throughput, and sharing resource.

The rest of this paper is organized as follows. Chapter 2 states the problems of resource management on a Web server. Chapter 3 introduces the architecture of the mQoS gateway and the designs of the request profiling and server profiling, content-aware request classification, and mQoS scheduling algorithm. Chapter 4 describes the implementation and evaluation of the mQoS gateway. Finally, Chapter 5 gives the conclusion and the future work of this research.

Chapter 2. Problems of Server Resource Management

The workload on a Web server will affect the utilization of the server resource. In the light-load situation, every HTTP request will get enough resources when being processed, but there could be unused resources on the server. Conversely, in the heavy-load situation, a request may be queued on the server and wait for being processed. If the server resources are inadequate for the requirements of the arrival requests, an HTTP request would experience long queuing and processing delay. For maximizing the utilization of the server resources and avoiding extra delay simultaneously, the resources on the server should be well managed.

Some researches have proposed admission control schemes to prevent new arrival requests from accessing a heavy loaded server[17-20]. With admission control, a server would drop new arrival requests when its resources cannot meet the requirements of the requests. However, admission control itself is not sufficient to support service differentiation because all arrival requests have the same probability to access server resources. The purpose of service differentiation is to allow different clients receive different treatments, such as server throughput and response time. For service differentiation, some researches have proposed request scheduling algorithms to control the workload on a server [1][2][18][21][22]. The general schemes of the mentioned scheduling algorithms are to allocate different amounts of concurrent connections, request rate, or bandwidth among service classes.

A request entering a server requires several types of resources, e.g. CPU, disk I/O, and bandwidth, when being processed. The lack of any available resource would lead to a bottleneck. In other words, if there are n kinds of resources, there could be n kinds of bottlenecks on the server. Many of the mentioned request scheduling

algorithms deal with the problems of single-resource scheduling. They manage a single resource for maximizing the utilization and differentiating the utilization simultaneously, but they cannot avoid the bottlenecks derived from the other resources. A resource can be managed well, while the other resources may be still available or inadequate for new arrival requests. A single-resource scheduling algorithm could lead to an inefficient or overloaded server. Hence, a request scheduling algorithm should consider the presence of multiple server resources. In the below, three requests scheduling schemes, no scheduling, single-resource request scheduling, and multiple-resource request scheduling, are discussed. The assumption for the discussion is that there are three resources, CPU, disk I/O, and bandwidth, on the server and a request will consume multiple resources. Besides, there are three service classes of clients issuing requests to the server, and the heavy-load situation is considered.

No Scheduling (nQoS)

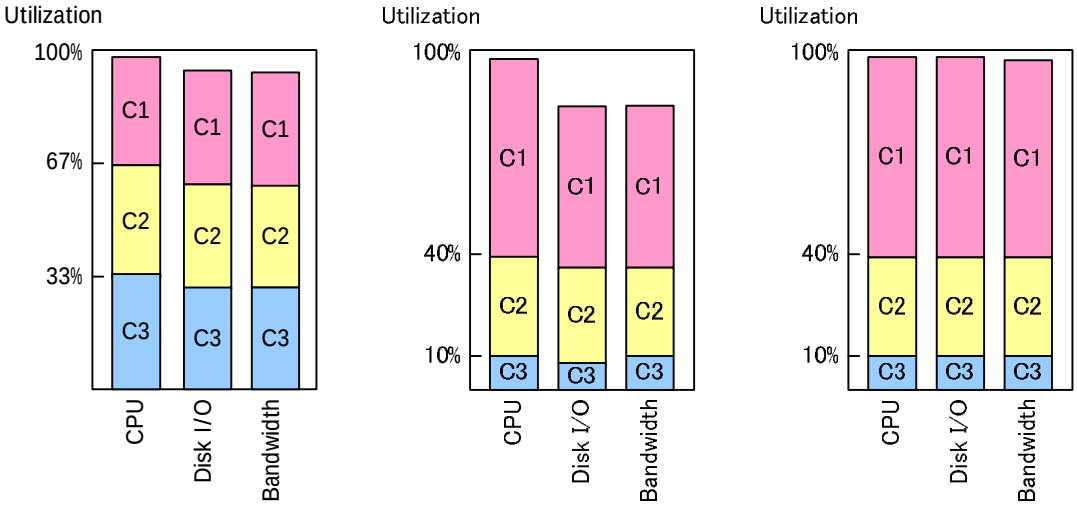
The nQoS scheduling is no any resource management scheme, such as admission control or request scheduling, enforces for the service differentiation. The requests originated from the three classes of clients contend for the server resources. The server works on a first-come-first-serve basis. The server workload of the nQoS scheduling is shown in Figure 1(a). The vertical axis stands for the resource utilization and c1, c2 and c3 stand for the class 1, class 2 and class 3, respectively. Due to the resource contention, every class of clients gets a third of each server resource. All the server resource utilization is effected by the workload, but there is no service differentiation. The pending requests would be queued on the server and wait for being processed, causing extra resource consumption, and prolonged user-perceived latency.

Single-resource Request Scheduling (sQoS)

In the sQoS scheduling, a request scheduler manages the utilization of one server resource. Figure 1(b) shows the server workload of the sQoS scheduling. The CPU resource is managed for service differentiation, and the ratio of the resource allocation to the three classes of clients is 6:3:1. In the example, the sQoS scheduling indeed allocates the expected amount of the CPU resource to the three classes of clients, but it cannot take care the utilization of the other resources. The sQoS scheduling will stop scheduling any requests to the server when the CPU resource is well utilized. However, the disk I/O and bandwidth resources are actually still affordable for the new arrival disk I/O- and bandwidth- intensive requests, respectively, causing the waste of disk I/O and bandwidth resources. Conversely, the sQoS scheduling will keep scheduling requests to the server when it finds the CPU resource is available. However, the disk I/O and bandwidth resources may be already fully utilized for the scheduled requests, causing an overloaded server and potentially prolonged user-perceived latency.

Multiple-resource Request Scheduling (mQoS)

In the mQoS scheduling, a request scheduler manages all the server resources. The server workload of the mQoS scheduling is shown in Figure 1(c). The mQoS scheduling chooses the appropriate requests to well utilize all the resources and at the same time allows the three classes of clients to use every resource proportionally. The mQoS scheduling eliminates the resource wasting or server overloading occurred in the sQoS scheduling, and the total server throughput can be improved. Due to scheduling the requests into the server, each resource utilization under mQoS is better than that under nQoS. The mQoS scheduling further avoids resource contention and enables service differentiation.



(a) nQoS scheduling (b) sQoS scheduling (c) mQoS scheduling

Figure 1. Server resource utilization under different scheduling schemes.

In the above discussion, the mQoS scheduling seems to be a better solution for server resource management. In this paper, a mQoS scheduling algorithm for service differentiation is presented. The mQoS scheduling algorithm has the capability of managing multiple server resources. The mQoS scheduling algorithm is deployed on a Website gateway located in front of a Web server. The arrival requests are queued and wait for being scheduled on the mQoS gateway instead of the server. This has the advantage of avoiding extra resource consumption on the server. The server itself can concentrate on the request processing only.

Chapter 3. The mQoS Gateway Architecture and Scheduling Algorithm

The purpose of the mQoS gateway is to avoid resource bottlenecks, provide differentiation of resource differentiation, and maximize the server throughput. To do this, the mQoS gateway performs three tasks: request profiling and server profiling, request classification, and request scheduling. The request profiling and server profiling let the mQoS gateway know the resource consumption of a request and the capacity of each server resource. The request classification allows the mQoS gateway to classify requests into different service classes. The request scheduling determines the order and the time in which the mQoS gateway sends a request to the server.

The architecture of the mQoS gateway, as shown in Figure 2, is composed of three components: server prober, request classifier, and request scheduler. The working flow of the gateway is described as follows. Before the on-line operation of the gateway, the server prober sends HTTP requests one by one to scan all the Web pages on the server. The resource monitor program running on the server monitors the resource consumption for every request and reports this information to the server prober. The server prober records the URLs and resource consumption of the Web pages in the Web page table for the reference of the request classifier. The QoS policy table defines the service classes and their classification rules. Once the gateway starts to work, it incepts arrival requests. The request classifier classifies the incepted requests into different service classes according to the rules defined in the QoS policy table. Then the request classifier refers to the Web page table, tags the information of the resource consumption to each request, and puts the tagged requests into the corresponding queues. The request scheduler checks the availability of the server

resources. If the available server resources are enough, the request scheduler fetches a request from a proper queue and sends it to the server. The detailed design of the server prober, request classifier, and request scheduler are discussed below.

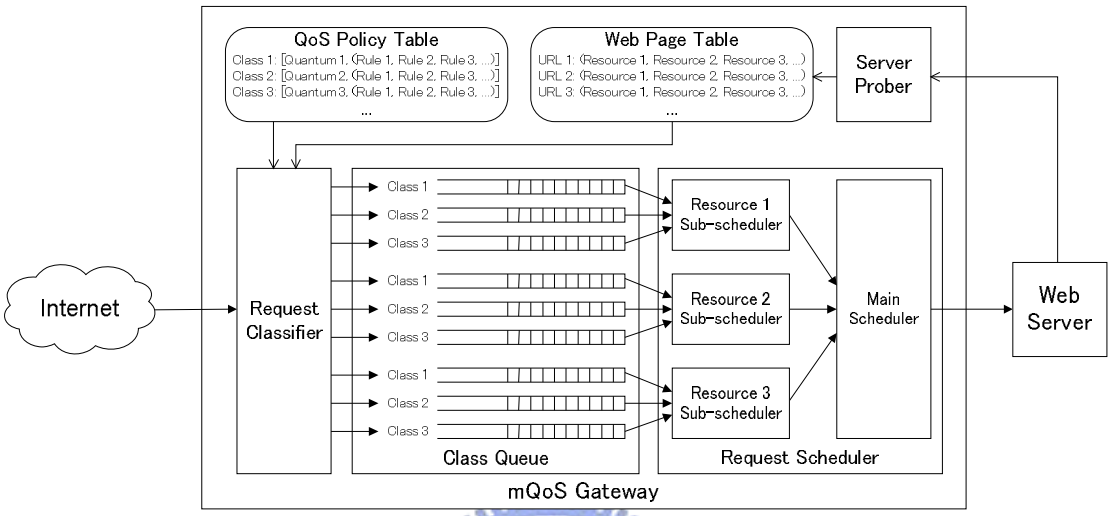


Figure 2. Architecture of the mQoS gateway.

3.1 Server Prober

The mQoS gateway is deployed in front of any type of Web servers. The gateway has to know the server resource consumption of a request and the capacity of each server resource for the management task. For this, the server prober is used for request profiling and server profiling. The request profiling is the process of measuring the resource consumption of a request, whereas the server profiling is the process of measuring the maximum capacity of each server resource.

For measuring the resource consumption of a request, the server prober sends HTTP requests one by one to scan all the Web pages on the server. Starting from the homepage, the server prober recursively parses every Web page and finds the URLs of the embedded objects and hyperlinks until the Web site is traversed. During the traversing, the monitor program running on the server monitors the amounts of server resources consumed for each request and reports this information to the server prober.

As an example, a query page consumes 15 units of CPU, 5 units of disk I/O and 8 units of bandwidth per second. To increase the validity of the measurement, the probed results are verified through the liner verification. That is, when the prober sends multiple requests to the server concurrently, the amount of the resource consumption is multiplied as the number of concurrent requests being processed on the server. Notice that this information is not directly used by the request scheduling algorithm because the actual percentage of the resource consumption is not known yet.

In order to calculate the percentage of the resource consumption of a request, the server prober has to measure the maximum capacity of each server resource. Thus, the server prober sends huge amount of specific resource-intensive requests at the same time to the server and checks the resource utilization. The maximum capacity can be measured when the resource is fully utilized. After all the resource capacities are measured, the actual capacities of the server resources and the percentages of the resource consumption of a request are derived. The maximum capacity of a server resource can be derived from multiplying the number of the concurrent requests on the server by the resource consumption of a request. As an example of measuring the CPU capacity, if there is 100 requests being processed by a fully-loaded server and the CPU resource consumption of each request is 15 units, then the maximum CPU capacity is 1500 units. The percentage of the CPU resource consumption of a request can be also derived from dividing its CPU resource consumption by the CPU capacity. In the above example of a query, its percentage of the CPU resource consumption is 1% (15/1500). The server prober finally records the URLs and resource consumption information in the Web page table for the use of the request classifier and request scheduler.

3.2 Content-aware Request Classifier

The request classifier is used to identify requests which classes and which resource tendency they belong. The classification is based on the predefined rules in the QoS policy table. The header and payload of a request will be inspected by the request classifier to check whether it matches a rule. If yes, the request will be classified into this corresponding class; otherwise, it will be compared with the other rules until classified. Once a request is classified, its URL will be inspected to match the URLs in the Web page table. The purpose is to find out the expected resource consumption and judge the tendency of the resource consumption. For example, a request consuming 9 % of CPU, 5 % of disk I/O and 7 % of bandwidth is regarded as a CPU-intensive request. After a request is matched with the QoS policy table and Web page table, the request classifier tags the information of the resource consumption to the request and put the request into an appropriate queue. Every service class has several queues, each of which stores specific resource-intensive requests. If there are m service classes and n server resources, there are totally m*n queues. The requests wait in the queues for being scheduled by the request scheduler.

3.3 Multiple-resource Request Scheduler

The request scheduler schedules the requests in the class queues to manage the server resources in order to provide service differentiation. The key idea of the mQoS scheduling is derived from the deficit round robin (DRR) scheduling for packet scheduling. A traditional DRR scheduler serves head-of-line (HOL) packet of every non-empty queue which the value of the deficit counter is greater than the size of the packet. If it is lower, then later the deficit counter is incremented by a given value called quantum. A deficit counter is decremented by the size of a packet. However, some considerations should be noticed on scheduling requests using the concept of the

DRR scheduling. The traditional DRR schedules packets to manage the bandwidth of a link, whereas the presented mQoS scheduler schedules requests to manage the multiple resources of a server. The utilization of the server resources has to be balanced. None of the resources should be overused or underused; otherwise a resource bottleneck would happen or a server resource would be wasted.

The mQoS scheduler consists of a ***main scheduler*** and several ***sub-schedulers***, as shown in Figure 3. A sub-scheduler services the class queues of a server resource for differentiating the resource utilization among the classes, and the main scheduler triggers an appropriate sub-scheduler according to the availability of the server resources. In a sub-scheduler, there are several deficit counters (DCs), each of which is associated with a class to record the unused quantum. However, differing from the traditional DRR scheduling, a DC of a sub-scheduler can be decremented by any other sub-schedulers because a request would consume multiple resources rather than single resource. Each sub-scheduler has a round-robin pointer that indicates which class queue to be serviced. When the round-robin pointer moves back to the first class queue, every DC of the sub-scheduler is incremented by the predefined quantum.

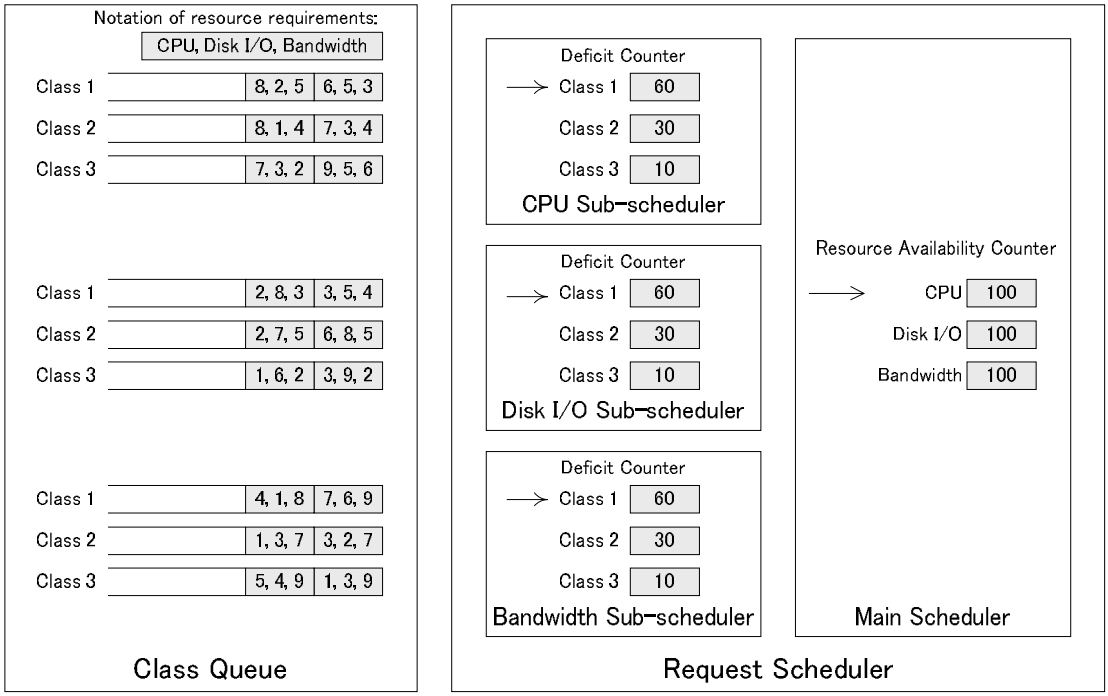


Figure 3. mQoS scheduler.

In the main scheduler, resource availability counters (RACs) are used to record the availability of the server resources. Each RAC contains the percentage of the availability of a server resource. By checking the RACs, the main scheduler knows which resource is the most available and then triggers the corresponding sub-scheduler to service a specific resource-intensive request. Therefore, the main scheduler can maximize the resource utilization and balance the utilization among the resources.

3.4 Multiple-Resource Request Scheduling Algorithm

The mQoS scheduling algorithm works as shown in Figure 4. Initially, the value of each RAC is set to 100, which means each type of server resource is 100% available. Each round-robin pointer in these sub-schedulers moves to the first class queue. In the traditional DRR scheduling, a DC is incremented only when the round-robin pointer moves to its corresponding queue. However, here all DCs of a

sub-scheduler are incremented at the same time by the predefined quantum because the DC of a sub-scheduler could be decremented by another sub-scheduler. The main scheduler checks the values of the RACs to find out which resource is the most available. A sub-scheduler will be triggered for scheduling the corresponding resource-intensive requests to effectively utilize the most available resource. The main scheduler randomly triggers a sub-scheduler, since there is no resource more available than the others.

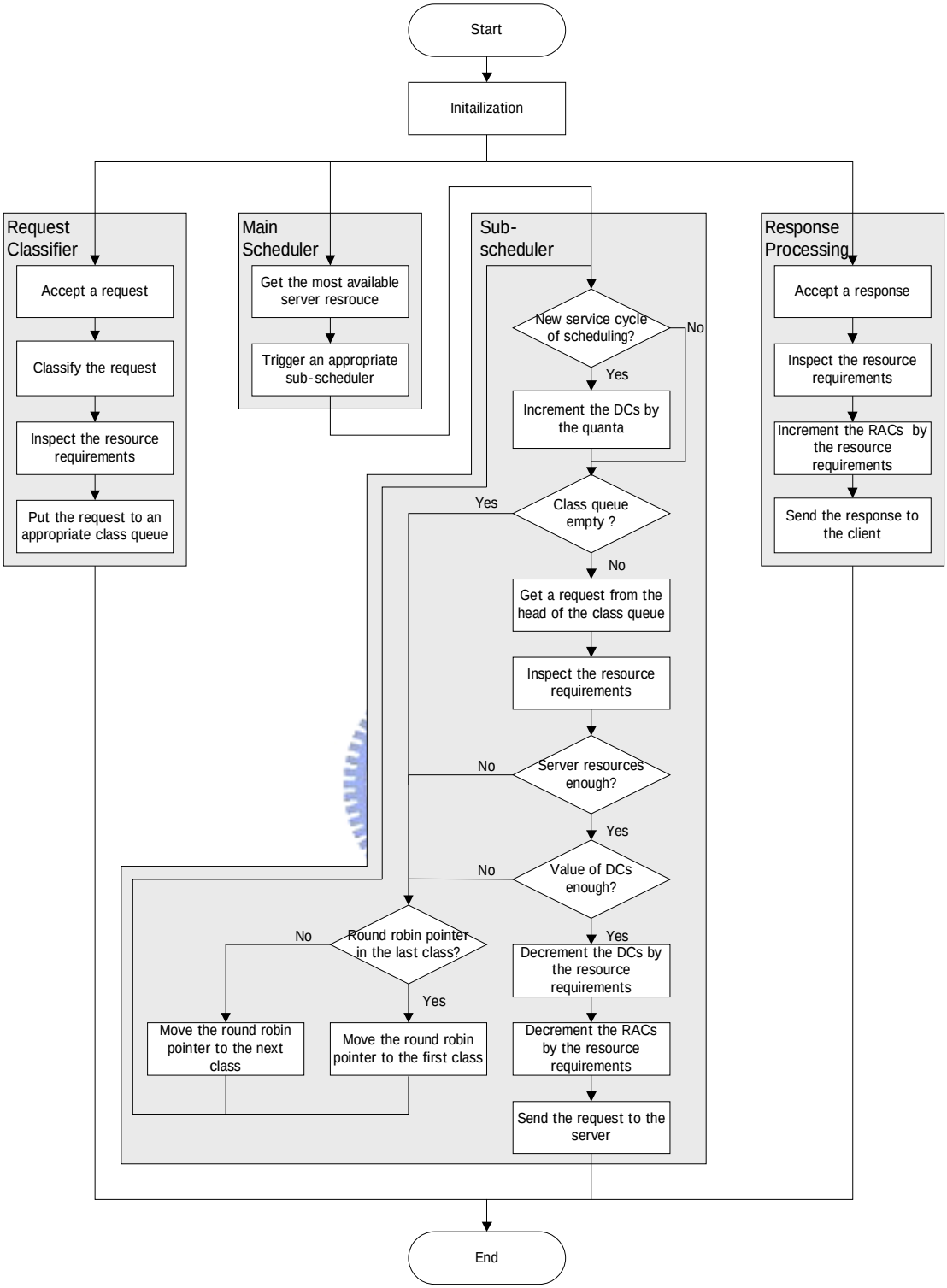


Figure 4. Flowchart of the mQoS scheduling algorithm.

The triggered sub-scheduler inspects the resource consumption information of the HOL request of the queue which the round-robin pointer locates. If no request waits in this queue, the sub-scheduler moves the round-robin pointer to the next queue

and the remaining deficit will not carried over to the next service cycle in the DC. The resource requirements of the request are compared with the values of the RACs. If any resource is not enough for the requirements, the sub-scheduler will move the round-robin pointer to the next queue without scheduling this request. If the resource requirements are satisfied, the sub-scheduler will check the values of the DCs of the same class from all the sub-schedulers to see whether this class has enough value in DCs. If no, the sub-scheduler will move the round-robin pointer to the next queue without scheduling the request. If yes, the sub-scheduler fetches the request from the queue, decrements the amounts of the resource requirements from the DCs and RACs, and sends this request to the server.

When the response from the server is back, the RACs will be incremented by the amounts of the resource requirements from the request to reflect the server releases the consumed resources. The main scheduler continues to trigger a sub-scheduler. A sub-scheduler continues to serve the requests from a queue until the queue becomes empty, the resource requirements cannot be satisfied. Since the scheduler has to be aware of the responses, the mQoS scheduler is not proper to work with direct routing.

The pseudo code of the mQoS scheduling algorithm is shown in Figure 5. The enqueuing module performs the request classification to put a request into an appropriate queue. The dequeuing module executes the mQoS scheduling algorithm to schedule the requests in the class queues. The response module checks the finish of a response and increments the RACs.

```
/* The definitions of variables
r: number of resources
c: number of classes
k: resource tendency
DC: deficit counter
RAC: resource availability counter
RRP: round-robin pointer
RR: resource requirement
```

Q: quantum
SF: service flag
QF: quantum flag
RF: service cycle flag
EF: empty flag
*/

Initialization:
For (i = 0; i < r; i = i + 1)
    RAC_i = 100; /* initialize resource availability counters */
    RF_i = TRUE; /* initialize service cycle flags */
    move_pointer(RRP_i, 0); /* move every round-robin pointer to the first class */
    For (j = 0; j < c, j = j + 1)
        DC_{ij} = 0; /* initialize deficit counters */
        QF_{ij} == TRUE; /* initialize quantum flags */
        EF_{ij} == FALSE; /* reset empty flags */

Request enqueuing module: on arrival of request p
j = get_class(p);
RR = get_resource_requirements(p);
k = get_resource_tendency(*RR*);
enqueue($Queue_{kj}$, p)

Request dequeuing modules:
While (TRUE)
    m = get_most_available_resource(*RAC*);

    RF_m = TURE;
    For (j=0; j < c ; j = j + 1) /* check a new service cycle */
        If (QF_{mj} == FALSE)
            RF_m = FALSE;

    If (RF_m == TRUE) /* increment deficit counters if it is a new service cycle */
        For (j = 0; j < c; j = j + 1)
            If(EF_{mj} == TRUE)
                DC_{mj} = 0;
                EF_{mj} = FALSE;
            DC_{mj} = DC_{mj} + Q_j; /* increment all deficit counters by quanta */
            QF_{mj} = FALSE;

    j = get_pointer_value(m);
    if((p = get_head_request($Queue_{mj}$)) == NULL)
        QF_{mj} = TRUE; /* ready to increment deficit counter by quantum in the next service cycle */
        EF_{mj} = TRUE; /* this is an empty case */

    RR = get_resource_requirements(p);

    SF = TRUE;
    For (i=0; i < r ; i = i + 1) /* check every resource */
        If (RAC_i < RR_i) or (DC_{ij} < RR_i) **then** /* meet resource requirements and deficits*/
            SF = FALSE;
            If (j == c - 1) then
                move_pointer(RRP_m, 0); /* move round-robin pointer to the first class */
            Else
                move_pointer(RRP_m, j + 1); /* move round-robin pointer to the next class */
            If (DC_{ij} < RR_i) **then**
                QF_{ij} = TRUE; /* ready to increment deficit counter by quantum in the next cycle */

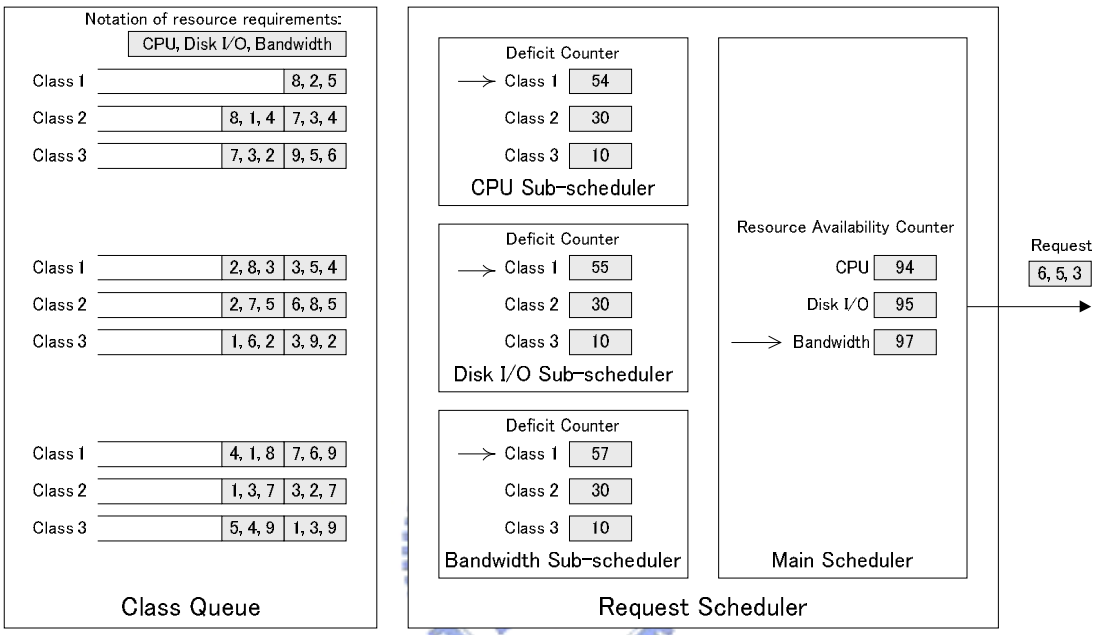
```
    If (SF == TRUE) /* service the request out */
        For (i=0; i < r ; i = i + 1)
            DC_ij = DC_ij - RR_i; /* decrement deficit counters */
            RAC_i = RAC_i - RR_i; /* decrement resource availability counters */
        send_request(p);

Response processing module: on arrival of response q
For (i=0; i < r ; i = i + 1)
    RR = get_resource_requirements(q);
    RAC_i = RAC_i + RR_i; /* increment resource availability counters */
send_response(q);
```

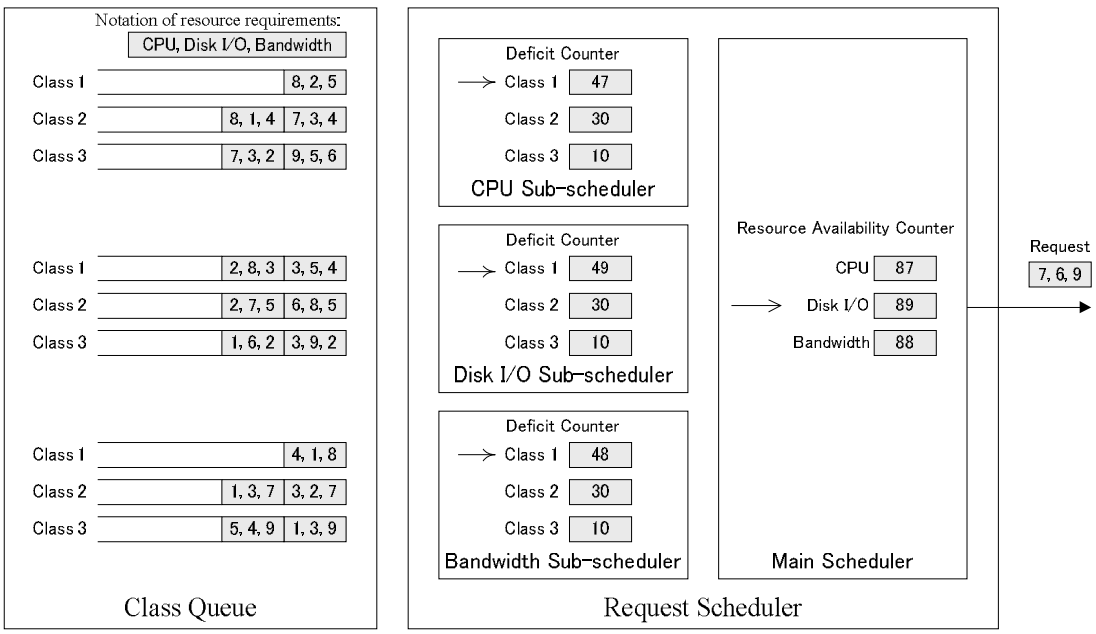
Figure 5. Pseudo Code of the mQoS scheduling algorithm.

Figure 6 exhibits an example of the mQoS scheduling. In this example, the requests are classed into three service classes: class 1, class 2, and class 3. The ratio of the service weights of the classes is set to 6:3:1, hence the quantum assigned to each class is 60, 30 and 10, respectively. The server resources to be managed are CPU, disk I/O, and bandwidth. Because there are three service classes and three server resources, totally nine class queues exist. The initial stage is shown in Figure 3. The main schedule randomly triggers the CPU sub-scheduler. The CPU sub-scheduler inspects the HOL request of the class-1 queue and knows the resource requirements of this request is (CPU: 6, disk I/O: 5, bandwidth: 3). The CPU sub-scheduler compares the amounts of the resource requirements to the values of the RACs (CPU: 100, disk I/O: 100, bandwidth: 100) and concludes the server resources are enough. Then it compares the resource requirements to the values of the DCs of the CPU, disk I/O, and bandwidth for class 1 (CPU: 60, disk I/O: 60, bandwidth: 60) and concludes the value in DCs is enough. The CPU sub-scheduler now sends the request to the server and decrements the DCs and RACs. The results of the decrements on the DCs and RACs are shown in Figure 6(a). The main scheduler now triggers the bandwidth sub-scheduler because the bandwidth resource is the most available. The bandwidth sub-scheduler sends the HOL request of the class-1 queue to the server. The result after this request scheduling is shown in Figure 6(b). Now the disk I/O resource

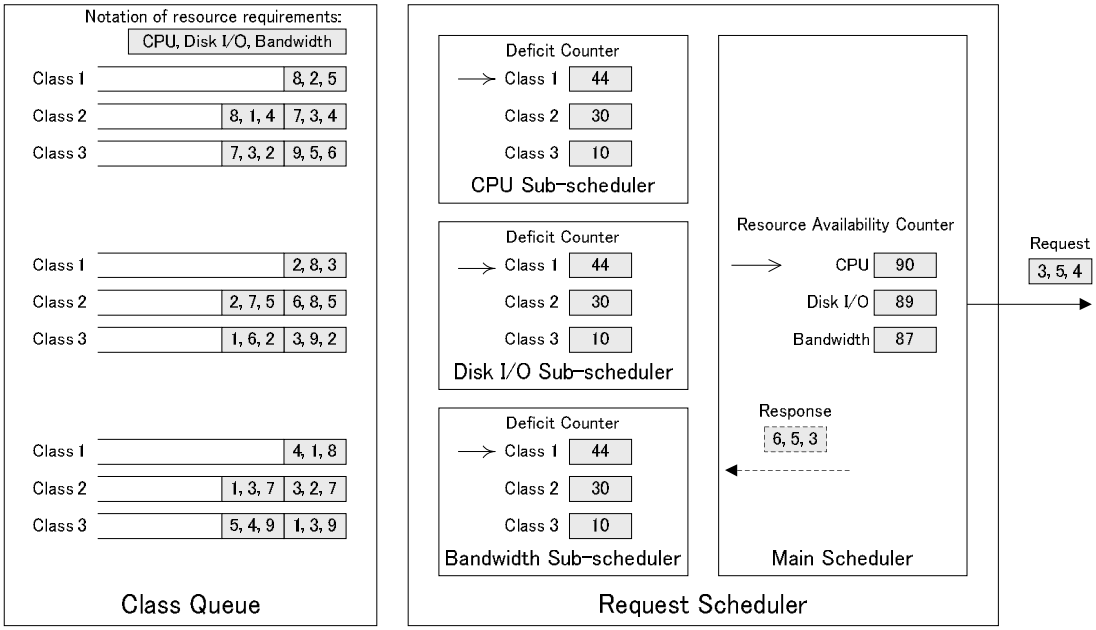
becomes the most available, hence the main scheduler triggers the disk I/O sub-scheduler to send a request. Suppose the server has finished responding the first request after the request sent by the disk I/O sub-scheduler. The final values of the RACs and DCs are shown in Figure 6(c).



(a) The CPU sub-scheduler sends a request.



(b) The bandwidth sub-scheduler sends a request.



(c) The disk I/O sub-scheduler sends a request and then a response returns.

Figure 6. An example of the mQoS scheduling.

Chapter 4. Implementation and Evaluation

4.1 Implementation

The implementation of the mQoS gateway is based on the Squid package and Linux operating system. The Squid package is modified to be capable of request classification and request scheduling. The Squid is of a single-process event-driven architecture, which uses the select() system call to simultaneously wait for events on all connections being handled. When select() delivers one or more events, the main loop of the Squid invokes handlers for each ready connection. The performance and scalability of the mQoS gateway is good because it does not need to fork a child process for each request. The server prober and resource monitor program are implemented as the server daemons running on the gateway and server, respectively. When a request enters the gateway, the iptable utility rewrites the destination IP address and port number of this incoming packet to redirect it to Squid. Such redirection mechanism makes the mQoS gateway works transparently to clients and the server. The Squid gateway performs request classification and scheduling and sends the request to the server. The Squid gateway then receives the response from the server without caching the response and sends it to the client.

The original Squid is a caching proxy used to cache the responses from a Web server. It is deployed between clients and servers to incept requests and responses. The request and response processing of Squid are shown in Figure 7. When a client issues a request, Squid reads the request, parses the request, and checks whether the response of this request is already in the cache. If yes, Squid fetches the cached data from the cache and sends it to the client. Otherwise, Squid prepares to forward the request and sends the request to the server. When the server returns a response, Squid reads the response, parses the response, and stores or replaces the response data in the

cache. Squid then prepares to forward the response and sends the response to the client.

In the mQoS gateway, the request and response processing modules of the Squid are modified to be capable of request classification and request scheduling. The cache module of checking in the request direction and the module of cache storing or replacing in the response direction are bypassed. Instead, the request classification is performed before Squid prepares to forward a request. Afterward, the request scheduling is performed before Squid sends a request to the server. When Squid finishes reading and parsing a response, the request scheduler updates the resource availability counters and then makes a preparation of forwarding the response to the client.

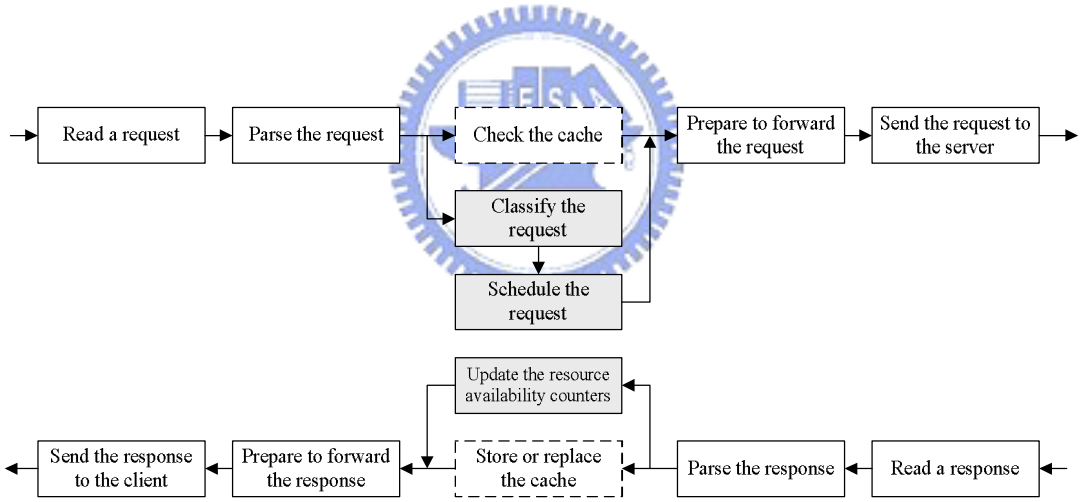


Figure 7. Implementation of the mQoS gateway on the Squid.

4.2 Evaluation

The effect of server resource management is discussed theoretically in Chapter 2. Here the implementations of the nQoS, sQoS, and mQoS scheduling are practically evaluated on server resource utilization, server throughput, and user-perceived latency. The evaluation environment consists of a traffic generator, a gateway, and a Web server, shown in Figure 8. Spirent's Avalanche software and SmartBits platform are

used as the traffic generator. Avalanche emulates a large number of clients to issue HTTP requests to the server and gather the statistics. The gateway performs the traditional DRR scheduling to manage the CPU resource of the server for the sQoS scheduling, or the mQoS scheduling algorithm to manage the CPU, disk I/O, and bandwidth resources. In the nQoS scheduling, the gateway only forwards requests and responses between the traffic generator and the server without any processing. The Web server is based on Apache and PHP.

There are three kinds of pages in the server, and different pages will lead to different consumption of the multiple resources when being accessed. The mathematic computation web pages compute equations and are CPU-intensive. The album web pages access photograph database and are Disk I/O-intensive. The chemical formula displaying web pages parse the formula notation to show the formula in 3D and are Bandwidth-intensive. In the evaluation, three service classes are defined in the QoS policy table, and the ratio of the quantum is set to 6:3:1. The workload contains three kinds of resource intensive requests, but the traffic generator issues more CPU-intensive requests than the other types of requests in order to test the capabilities of the mQoS scheduling. The server as 640 MHz CPU and 128 MB RAM, and the gateway has 700 MHz and 256 MB RAM. Avalanche keeps 600 outstanding requests from clients.

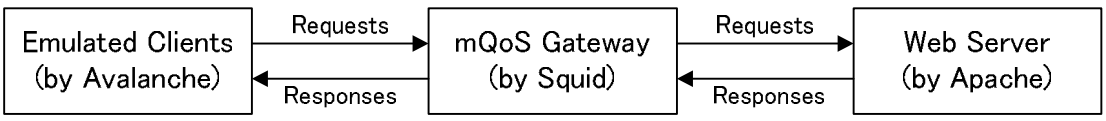
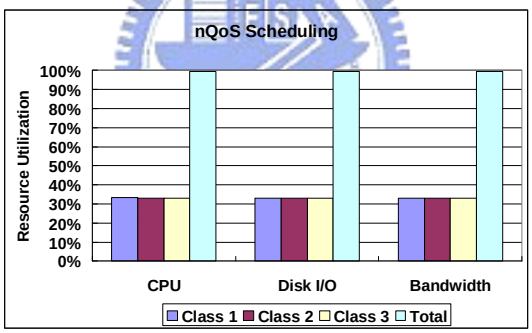


Figure 8. Evaluation environment

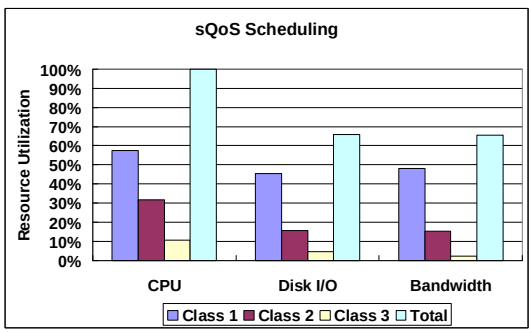
4.3 Differentiation on the Resource Utilization

Different request scheduling schemes result in different utilization of the server resources, shown in Figure 9. In Figure 9(a), the nQoS scheduling, every class gets a

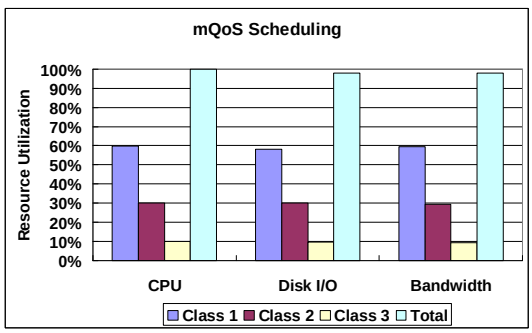
third of every server resource due to the resource contention. Although three resources are well utilized, there is no differentiation on the resource utilization among three classes. In Figure 9(b), the sQoS scheduling, the gateway schedules requests to well utilize the CPU resource of the server and simultaneously to differentiate the resource utilization to the ratio of 6:3:1. However, the gateway stops sending requests to the server when the CPU resource of the server is well utilized, causing the waste of the disk I/O and bandwidth resources of the server. In Figure 9(c), the mQoS scheduler sends appropriate requests to the server to well utilize the three server resources. Furthermore, the differentiation of the resource utilization is evidently observed from that every server resource is utilized by the three classes according to the defined ratio of 6:3:1.



(a) Resource utilization in the nQoS scheduling.



(b) Resource utilization in the sQoS scheduling.



(c) Resource utilization in the mQoS scheduling.

Figure 9. Server resource utilization of the nQoS, sQoS, and mQoS scheduling.

4.4 Differentiation on the Server Throughput

The amount of the utilization of every server resource will affect the server throughput, as presented in Figure 10. In the nQoS and mQoS scheduling, the maximum total throughput is close to 300 requests per second which is limited by the server throughput. However, in the sQoS scheduling, due to the waste of the disk I/O and bandwidth resources of the server, the total throughput is only 260 requests per second. The mQoS scheduling improves the total throughput by 21% from sQoS. Another finding is that there is no differentiation on the server throughput among the three classes in the nQoS scheduling. However, the sQoS and mQoS scheduling reveal the differentiation on the server throughput because they schedule requests for different classes. The ratio of the server throughput of the three classes is close to 6:3:1.

In Figure 10, the server throughput of the nQoS scheduling is close to that of the mQoS scheduling. The server service rate is within limited, because of the limited server resource. The workload in the nQoS and mQoS scheduling make the server resource well utilized. In nQoS case, the server faces uncontrolled heavy request arrival rate, whereas in mQoS case, the server faces the scheduled request arrival rate which can well utilize the server. Although under these different situations, the server throughput is still limited by the server service rate. Due to uncontrolled request arrival rate, the user-perceived latency in nQoS is longer than that in mQoS. Figure 13 provides an evidence of this.

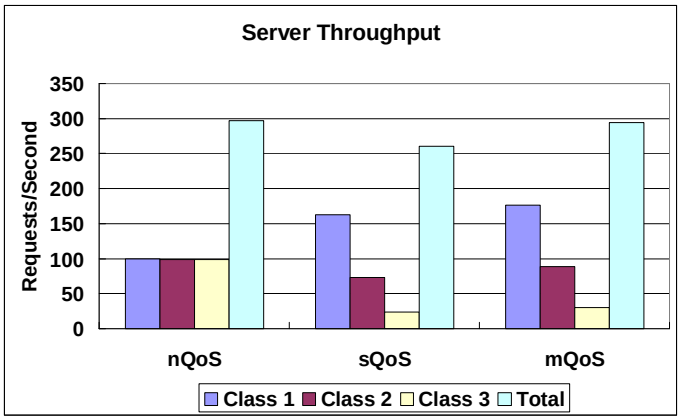
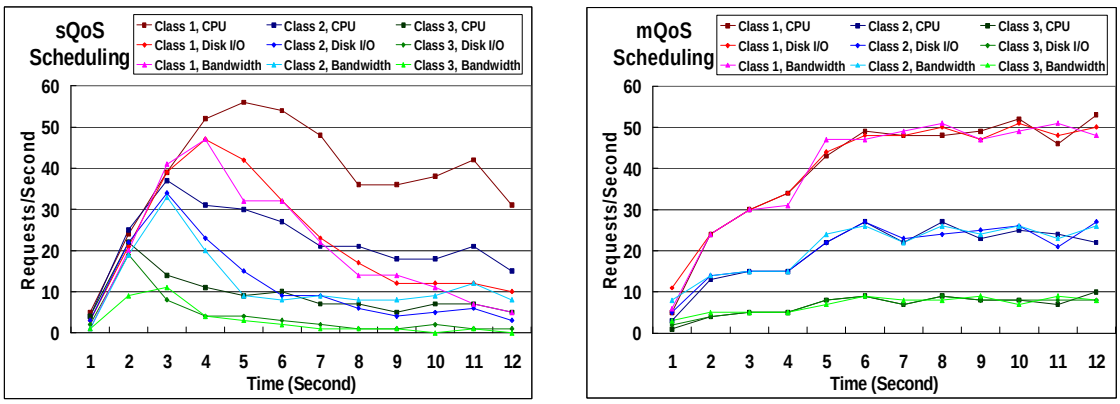


Figure 10. Server throughputs of the nQoS, sQoS, and mQoS scheduling.

The throughput improvement in the mQoS scheduling results from the fact that the gateway sends appropriate requests to the server to effectively utilize the three server resources. Figure 11 compares the types of outstanding requests between the sQoS and mQoS scheduling. In Figure 11(a), the sQoS scheduling, the gateway does not try to balance the utilization of the server resources. However in Figure 11(b), the mQoS scheduling, the main scheduler takes effect to balance the utilization on every resource. Also the three sub-schedulers differentiate the utilization of every resource among the three classes with a ratio close to 6:3:1.



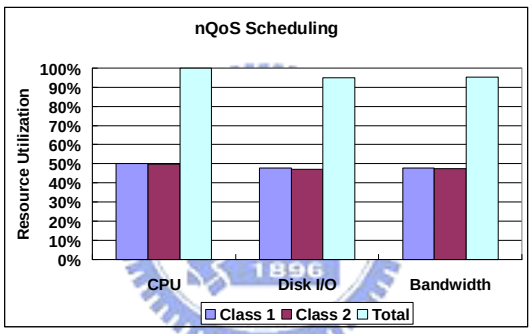
(a) Types of outstanding requests sent by the sQoS scheduling

(b) Types of outstanding requests sent by the mQoS scheduling

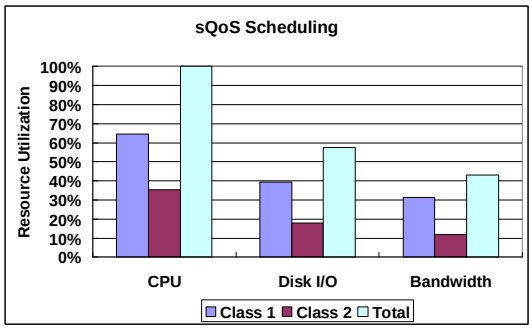
Figure 11. Types of outstanding requests by the sQoS and mQoS scheduling.

4.5 Resource Sharing

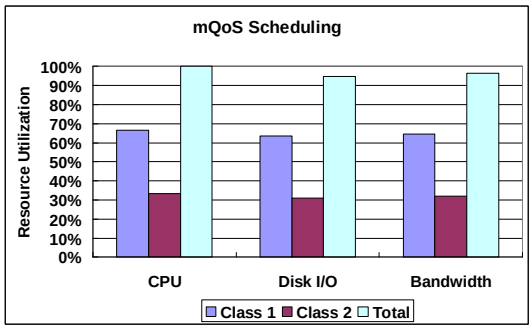
In the evaluation of the differentiation on the resource utilization, the three classes proportion the utilization of three server resources. If there are no request for a class, the available resources should be shared among the active classes (i.e., those having requests in the queues) in proportion. Figure 12 shows the situation of the resource sharing. The clients of the class 3 stop sending requests to the server during the evaluation. The server resources are then shared by the clients of the class 1 and class 2. The observation of Figure 12 is similar to that of Figure 9. In the mQoS scheduling, every server resource is utilized by the two active classes according the ratio of 6:3.



(a) Resource sharing in the nQoS scheduling.



(b) Resource sharing in the sQoS scheduling.



(c) Resource sharing in the mQoS scheduling.

Figure 12. Resource sharing of the nQoS, sQoS, and mQoS scheduling.

4.6 Differentiation on the User-perceived Latency

User-perceived latency is the time between issuing a request and receiving a response back at the client. Figure 13 shows the user-perceived latency of the nQoS,

sQoS, and mQoS scheduling. For the nQoS scheduling, there is no differentiation on the user-perceived latency among the three classes. Because the heavy workload leads to requests queued on the server, the average latency is longer than mQoS scheduling. For the sQoS scheduling, although the user-perceived latency is differentiated, the average latency is longer. For the mQoS scheduling, even though the mQoS gateway differentiates the server resources, the user-perceived latency is also differentiated but the ratio is not exactly 6:3:1. Furthermore, the average user-perceived latency of the mQoS scheduling is shorter than those of the nQoS and sQoS scheduling.

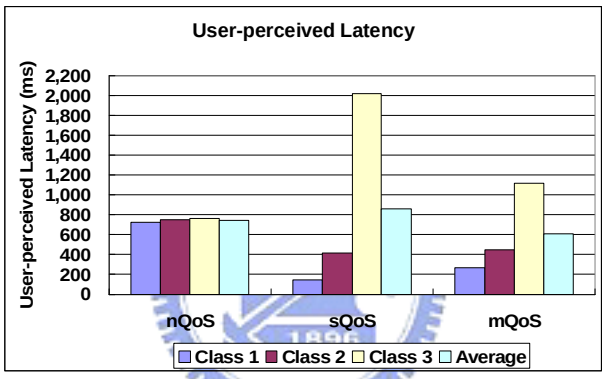


Figure 13. User-perceived latency of the nQoS, sQoS, and mQoS scheduling.

Decomposition of the User-Perceived Latency

The user-perceived latency in the mQoS scheduling mainly consists of the gateway queuing time and server processing time. The gateway queuing time is the time between accepting a request from the client and scheduling the request to the server at the gateway. The server processing time is the time between accepting a request from the gateway and sending the response to the client at the server. Figure 14 shows the decomposition of the user-perceived latency in the mQoS scheduling. The server processing time is almost the same among the three classes, whereas the queuing time of every class is different. Different queuing times lead to the

differentiation on the user-perceived latency.

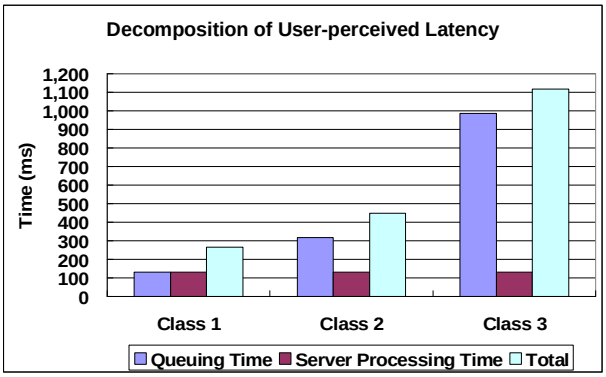


Figure 14. Decomposition of the user-perceived latency in the mQoS scheduling.

Chapter 5. Conclusion and Future Work

Resource management on a Web server allows a Website operator to control the utilization of the server resource and provide differentiated quality of service. Traditional single-resource request scheduling cannot manage multiple server resources well, that leads to resource wasting or overloading. This research presents a multiple-resource request scheduling algorithm, call mQoS, deployed at the Website gateway to provide service differentiation. The mQoS gateway consists of a server prober, a request classifier, and a request scheduler. The server prober profiles the resource consumption of every Web page and the capacity of every server resource. The content-aware request classifier determines the resource tendency and the service class of requests to classify them into different class queues. The mQoS scheduler consists of several sub-schedulers and a main scheduler. Each sub-scheduler manages a server resource and differentiates the resource utilization among the classes. The main scheduler checks the availability of the server resources and triggers an appropriate sub-scheduler to balance the utilization among the resources. The mQoS scheduling algorithm is work-conservative to the server to keep the server resources well utilized. However, it is non-work-conservative to the class queues because the scheduler remains idle when there are no enough resources for servicing a request.

The mQoS gateway is implemented on the Squid and Linux. The mQoS scheduling algorithm is compared with no scheduling (nQoS) and single-resource request scheduling (sQoS). The nQoS exposes no differentiation, and the sQoS exposes the differentiation only on the utilization of one server resource. However, the mQoS scheduling reveals the differentiation on the utilization of every server resource. Because all server resources are well utilized in the mQoS scheduling, the total server throughput is improved by 21%, compared with the sQoS scheduling. Moreover, the

user-perceived latency is also differentiated among the classes in the mQoS scheduling due to the differentiation of the gateway queuing delay. In the evaluation, the mQoS scheduling reveals its capabilities of differentiating the server resource utilization, maximizing the server throughput, and sharing resource.

The presented mQoS scheduling algorithm is for one Web server. It should be improved to support scheduling requests for a cluster of servers. The more complex multiple-resource, multiple-server request scheduling algorithm can be implemented on a server load balancer. The issues of service differentiation, resource utilization, and server load balancing should be completely considered in the design of the new algorithm.

References

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao, "Providing Differentiated Levels of Service in Web Content Hosting," *Proceedings of the 1st Workshop Internet Server Performance*, Jun. 1998.

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

[3] R. Pandey, J. F. Barnes, and R. Olsson, "Supporting Quality of Service in HTTP Servers," *Proceedings of the 7th Annul ACM Symposium on Principles of Distributed Computing*, pp. 247-256, Jun. 1998.

[4] V. Cardellini, E. Casalicchio, M. Colajanni, and M. Mambelli, "Web Switch Support for Differentiated Services," *ACM Performance Evaluation Review*, vol. 29, no. 2, pp. 14-19, Sep. 2001.

[5] H. Zhu, H. Tang, and T. Yang, "Demand-driven Service Differentiation in Cluster-based Network Servers," *Proceedings of the 20th Conference of the IEEE Communications Society*, vol. 2, pp. 679-688, Apr. 2001.

[6] K. Shen, H. Tang, and T. Yang, "A Flexible QoS Framework for Cluster-based Network Services," *Proceedings of the 2002 USENIX Annual Technical Conference*, Dec. 2002.

[7] S. Chandra, C. S. Ellis, and A. Vahdat, "Application-Level Differentiated Multimedia Web Services Using Quality Aware Transcoding," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 12, Dec. 2000.

[8] C. C. Hung and L. Y. Hong, "Adaptive Proxy-based Content Transformation Framework for the World-Wide Web," *Proceedings of the 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific*

Region, vol.2, pp. 747-750, May 2000.

[9] 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 10th International Workshop on Quality of Service*, May 2002.

[10] W. Leinberger, G. Karypis, and V. Kumar, "Job Scheduling in the presence of Multiple Resource Requirements", *Proceedings of the 7th International Conference on High Performance Networking and Computing*, Apr.1999.

[11] W. Leinberger, G. Karypis, and V. Kumar, "Load Balancing Across Near-Homogeneous Multi-Resource Servers", *Proceedings of the 9th Heterogeneous Computing Workshop*, pp. 60-71, May 2000.

[12] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar and J. Hansen, "A Scalable Solution to the Multi-Resource QoS Problem," *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Dec. 1999.

[13] M. E. Crovella, R. Frangioso and M. Harchol-Balter, "Connection Scheduling in Web Servers," *Proceedings of the 1999 USENIX Symposium on Internet Technologies and System*, Oct. 1999.

[14] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers," *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 90-101, Jun. 2000.

[15] 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*, pp. 535-544, May 2001.

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

1996.

[17] X. Chen, P. Mohapatra, and H. Chen, "An Admission Control Scheme for Predictable Server Response Time for Web Accesses," *Proceedings of the 10th World Wide Web Conference*, pp. 545-554, May 2001.

[18] K. Li, and S. Jamin, "A Measure-Based Admission Control Web server," *Proceedings of the 9th Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 2, pp. 651-659, Mar. 2002.

[19] L. Cherkasova and P. Phaal, "Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites," *IEEE Transactions on Computer*s, vol. 51, issue 6, pp. 669-685, Jun. 2002.

[20] S. Elnikety, J. Treacy, E. Nahum, and W. Zwaenepol, "A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites," *Proceedings of the 13th International World Wide Web Conference*, pp. 276-286, May 2004.

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

[22] V. Kanodia and E. W. Knightyly, "Ensuring Latency Targets in Multiclass Web Servers," *IEEE Transaction on Parallel and Distributed Systems*, vol. 14, no. 1, pp. 84-93, Jan. 2003.