國立交通大學

資訊科學與工程研究所

碩士論文

在網站閘道器上提供整體回應時間比例差別服務之多重資源要求排程演算法

Multiple-resource Request Scheduling Algorithms for Proportional System-time Differentiation at Website Gateway

研 究 生：陳銘宏

指導教授：林盈達 教授

中華民國 九十五 年六月

在網站閘道器上提供整體回應時間比例差別服務之多重資源要求排

程演算法

Multiple-resource Request Scheduling Algorithms for Proportional System-time Differentiation at Website Gateway

研 究 生：陳銘宏 Student：Ming-Hung Chen

指導教授：林盈達 Advisor：Ying-Dar Lin

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年六月

在網站閘道器上提供整體回應時間比例差別服務之多重資源要求排程演算法

學生：陳銘宏　　　　　指導教授：林盈達

國立交通大學資訊工程研究所

摘 要

對於一個商業網站來說，如何最佳化網站伺服器的吞吐量與提供重要的客戶較短延遲時間的服務是兩個需要考量的主要問題。本篇論文提出了一個置於網站之前，透過多重資源要求排程演算法來提供不同等級的客戶間整體回應時間比例差別服務的閘道器系統。即使這個閘道器系統完全獨立於網站之外，此閘道器系統依然可以完全的消耗伺服器上的資源以提升網站吞吐量，並且還可以提供比例的整體回應時間差別服務給不同等級的使用者。這個閘道器系統主要由許可控制機制與要求排程機制兩大部分所組成，其中的許可控制機制透過控制轉送要求的速度，除了可以避免網站伺服器過載，還可以盡量的有效運用網站伺服器上的資源；而要求排程機制則依據三種修改過的比例延遲演算法, WTP, MDP 和 PAD 來進行要求排程，以提供不同等級間的整體回應時間比例差別服務。為了驗證效果，我們透過修改一個開放原始碼的代理伺服器軟體 Squid 來實做這個閘道器系統，測試結果中顯示此閘道器系統可以提升最大的吞吐量達 78%，並同時減少 25%的整體回應時間。除此之外，我們也發現以 WTP 和 MDP 為基礎的排程演算法可以提供良好的整體回應時間比例差別服務效果。

關鍵字：多重資源、請求排程、差別服務、延遲

Multiple-resource Request Scheduling Algorithms for Proportional System-time Differentiation at Website Gateway

Student: Ming-Hung Chen Advisor: Dr. Ying-Dar Lin

Department of Computer Science
National Chiao Tung University

Abstract

Optimizing the serving throughput and providing important customers short user-perceived latency are two of major concerns for commercial websites. This thesis proposes a gateway system in front of the website to provide Proportional System-time Differentiation with multiple-resources consideration (MR-PSTD) between customers. Despite being external to the website, our gateway not only exhausts the resources of all types in the website to raise its throughput, but also provides proportionally differential system time differentiation to users of different classes. The gateway mainly consists of an admission controller (AC) and a request scheduler (RS). To prevent the server from being overloaded while exhausting all resources, AC controls the forwarding rate of requests to the server. To provide proportionally differentiated system time between classes, RS schedules the requests according to three reformed latency-based scheduling algorithms including Waiting Time Priority (WTP), Mean Delay Proportional (MDP) and Proportional Average Delay (PAD). We implement our gateway by modifying Squid, an open-source proxy. Our evaluation results show that our MR-PSTD gateway raises 78% of peak throughput and reduces 25% of user-perceived latency. Besides, WTP-based and MDP-based scheduling algorithms can provide exact PSTD between classes.

Keywords: multiple resources, request scheduling, differentiation, latency

Contents

Chapter 1 Introduction 1
Chapter 2 Multiple-Resources PSTD 3
2.1 More Desirable PSTD than PDD and PSD 3
2.2 Multiple-resources consideration 4
Chapter 3 MR-PSTD Gateway Architecture 6
3.1 Overview of gateway architecture 6
3.2 Classification of classes and types 7
3.3 Admission control module 8
3.3.1 Offline measurement for server capacity 8
3.3.2 Decide the next type 11
3.4 Request scheduler 12
3.4.1 The reformed algorithms 13
3.5 Response handler 15
3.5.1 Updating the mean service time 16
3.5.2 Adjusting the upper-bound of window 16
Chapter 4 Experiment and Results 19
4.1 Software Implementation 19
4.2 Test Bed 21
4.3 Effects on Differentiation 23
4.3.1 WTP is stable in short timescales 23
4.3.2 PWAD is unsuitable for PSTD 25
4.3.3 MDP is stable in medium timescales 26
4.4 Performance Improvement 28
Chapter 5 Conclusion and Future Works 32
References 33

List of Figures

Figure 1 Resource utilization on server 5
Figure.2 The MR-PSTD Web Gateway Architecture 6
Figure 3 Procedure of offline measurement...11
Figure 5 Procedure of dynamic window size adjusting 18
Figure.6 The modified processing flow of Squid 20
Figure.7 Experiment network topology 22
Figure.7 Averaged system time of WTP-based MR-PSTD gateway 24
Figure 8 Averaged system time of PWAD based MR-PSTD gateway 26
Figure 9 Averaged system time of MDP based MR-PSTD gateway 27
Figure 10 Average throughputs between gateways..29
Figure 11 Throughput with different upper bound of service time............................30
Figure 12 Service time and queuing time distribution between gateways..................30
Figure 13 Number of concurrent requests between gateways 31

List of Tables

Table 1 The summary of reformed algorithms 13
Table 2 The description of new functions 21
Table 3 Software running on PCs 22
Table 4 Experiment results of offline probing 23

Chapter 1 Introduction

The user-perceived latency is a key metric for a web site to evaluate its providing service for the users. If the users perceive a short latency on visiting a web site, they may have a high willing to visit it frequently. For an e-commerce web site, it is particularly important that their customers perceive short latency because the customers would support the finances of the site. To provide short latency for customers, expending the server resources, e.g. buying more servers, is a simple strategy for web-site operators. However, expending resources costs money. Therefore, under the limited server resources, the operators may expect a solution that can (1) fully utilize all types of resources to provide the optimal user-perceived latency and (2) provide differential user-perceived latency for customers of different classes. That is, high-class customers can get a shorter latency than low-class ones.

The Proportional Delay Differentiation (PDD) model proposed in [3] aims to provide differential perceived latency for users of different classes. However, because the PDD model is proposed for network-side QoS where the service time of packets is short and ignored by the PDD model, the PDD model only considers the queuing time of packets on the router or gateway. In server-side QoS, the service time of requests may be long and should not be ignored anymore, and the PDD model is not suitable anymore. The user-perceived latency in server-side QoS must consist of the queuing time and the service time on the server or the web gateway simultaneously, and in this situation, the user-perceived latency is also called the system time.

Thus, to fully utilize all types of resources in the server and provide differential system time in server-side QoS, this work proposes a solution to satisfy the e-commerce web-site operators. Our solution is deployed in the proxy in front of the web sites, so the source codes at the web site do not need modification. This approach

is significantly different from the previous solutions [1-2], which runs at the server directly. Our solution includes two key modules: the admission control and request scheduling. The former controls the number of the requests which will be forwarded to the server and simultaneously served. The latter decides the request in which class would be forwarded next.

In our solution, to fully use all types of resources at the server, the requests are classified based on the major type of resources they need. For each type of requests, our admission control employs an individual window to control the number of these requests simultaneously served in the server. That is, the requests taking different resources would be forwarded independently and served simultaneously, because the time to simultaneously serve these requests should be shorter than that to sequentially serve. Besides, each window is dynamically adjusted based on the utilization on the corresponding type of resources. The utilizations are periodically probed from the proxy.

To provide differential user-perceived latency at proxy, the requests are online classified based on the customers who issued them. Each class of customers is assigned a weight and their requests will be classified into a corresponding queue. Next, our request scheduling selects requests from these class queues with a special order in order to ensure the Proportional System-Time Differentiation (PSTD) among different classes. For example, the requests in high-class with weight 2 will have half of system-time experienced by the lower-class ones with weight 1. The system time represents the total delay of a request, including the queuing time at the proxy and the service time at the web sites.

The organization of this work is explained as follows. Chapter 2 explains why the PSTD is more desirable than other previous models [2-3] and multiple-resources consideration is important. Chapter 3 presents the gateway architecture that can

achieve MR-PSTD. Chapter 4 presents the implementation of this gateway and the experimental results. Chapter 5 concludes this work.

Chapter 2 Multiple-Resources PSTD

The chapter explains (1) why PSTD is a more desirable goal for server-side QoS than two previous models: proportional delay differentiation (PDD) [3] and proportional slowdown differentiation (PSD) [2], and (2) why customers have shorter system time under multiple-resources consideration than under single-resource consideration. That is, why considering multiple resources can lead to shorter system time than merely considering single resource.

2.1 More Desirable PSTD than PDD and PSD

The following briefs the PDD and PSD models and describe why PSTD is more desirable for server-side QoS than PDD and PSD.

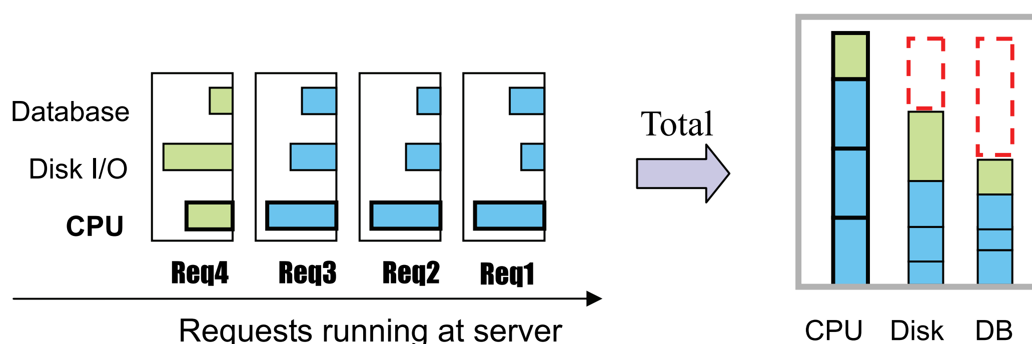
The PDD model was proposed to provide network-side QoS [3]. It describes a situation that the queuing delays among packets of different classes are proportional. PDD ignores the service time of the packet, i.e. the time to transmit the packet, because it is fixed and far smaller than the queuing time. However, under server-side QoS, ignoring the service time cannot guarantee the proportional differentiation on user-perceived latency because the service time is variable and long.

Compared with PDD, the PSD model is proposed to provide server-side QoS and consider the service time [2-3]. However, PSD may not satisfy the high-class customers because it plans to provide different classes with the proportional differentiation on *slowdown*, not directly on the user-perceived latency. The slowdown represents the quotient of dividing the queuing time by service time. High-class customers may always expect short latency for all their requests and do not care about

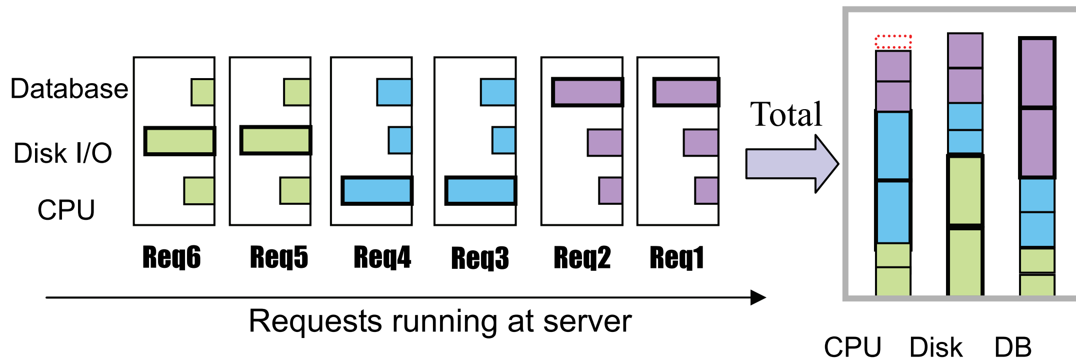
the length of the service time.

2.2 Multiple-resources consideration

As shown in Figure 1(a), because the server has multiple kinds of limited resources, when the server serves requests in a FIFO manner, one of the resources may be fully used while others are idle. In Figure 1(b), the resources on server are almost exhausted, because the difference of incoming requests. Thus, the pattern of workload on a Web server will affect the utilization of the server resource. When the server is under the light-load situation, every request will get enough resources when being served, but there could be unused resources on the server. The waste of idle resources may lead to long queuing time and low throughput on the gateway. Contrarily, under the heavy-load situation, a request may queue on the server and wait to be served for a long time. Besides, since many requests are serving simultaneously, the server would have frequent content-switching, which overhead may degrade the performance of the server. Thus, if the server resources are inadequate for the requirements of the arrival requests, a request would stay at the server for a long time. The performance of the server may degrade, and the system time of requests may increase much. To maximize the utilization of the server resources while avoiding extra delay, the resources on the server should be well managed.



(a) Waste of idle resources



(b) Fully used resources

Figure 1 Resource utilization on server

Serving a request requires several types of resources, e.g. CPU, disk I/O, and external database access. The unavailability of any resource would lead to a bottleneck, e.g. the server shown in Figure 1(a) where the CPU resource is the bottleneck. In other words, if there are *n* types of resources, there could be *n* types of bottlenecks on server-side.

Many of the mentioned request-scheduling algorithms deal with the problems of single-resource bottleneck [1-7]. They manage a single resource to simultaneously maximize and differentiate its utilization, 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 arriving 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 resources on server.

Chapter 3 MR-PSTD Gateway Architecture

This chapter proposes a web gateway architecture which can achieve the PSTD model with multiple-resources consideration (MR-PSTD). Two key modules: admission control and request scheduling, which respectively determine when to send the next request and how to select it, will be described in the following.

3.1 Overview of gateway architecture

Figure 2 is a typical network topology where a two-tier web site is serving requests received from Internet. The MR-PSTD web gateway locates in front of the web server. Requests sent from end users are forwarded by the gateway, served by the server, responded to the gateway, and finally returned to the end users. The gateway architecture, as shown in the bottom of Figure 2, includes several components: request classifier, request scheduler, admission control, service-time prober, response classifier, and response handler.

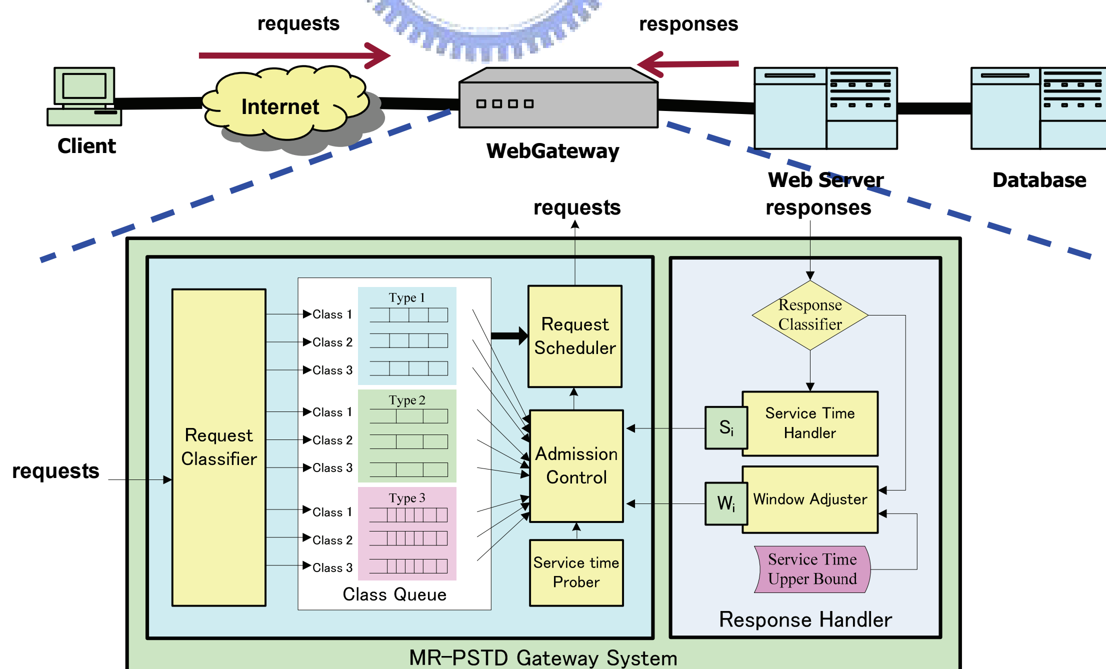


Figure.2 The MR-PSTD Web Gateway Architecture

The working flow of the MR-PSTD web gateway can be divided into three steps

and is described as follows. First, when a request comes to the gateway, the request classifier, which is a content-aware classifier, identifies the class and the resource-type of the request and puts it into the corresponding queue. In this architecture, there are $m * n$ of queues, where m denotes the number of classes and n denotes the number of types of resources. Second, when the admission control module, which uses window rate control mechanism to manage the sending rate of requests, regards that the server has enough resources to serve a type-i request, it asks the request scheduler to picks up a request from one of the queues that storing the type-i requests. The request scheduler must decide to pick up the request from which class of queue. Third, after the server sends responses back to the gateway, the content-aware response classifier identifies the type and class of responses. The response handler is invoked next and collects information to update averaged service time or adjust upper-bound of window if needed.

3.2 Classification of classes and types

To handle the multiple resources and provide differentiated service in the web site, we have to classify requests accord to their required resource types and customers' classes by scanning the content of requests. To classify requests based on the resource type, we first assume the web master knows the internal operations required by each request in the server. Then, according to the major type of resources required by the request, the master can classify them into the corresponding types, such as database operation intensive, math calculation intensive, and file access intensive.

To classify requests by the customer class, the request classifier has to recognize which class of customers issuing the request. The requests coming from the specific-class customers may bring the specific information, e.g. URL paths, or

cookies in their HTTP headers. Hence, we can classify the requests to the corresponding classes by these specific information.

Briefly, when a request arrives in the gateway, the content-aware request classifier recognizes the class and type of the request by its HTTP header and puts it into the individual queue corresponding to its class and type. The request is queued until the request scheduler selects it from the queue.

3.3 Admission control module

The Admission Control (AC) module decides when to release a request to the server. AC is necessary because the resources in the server are limited. Sending requests without any control causes the server overload and degrades the performance. To fully exhaust one type of resources, but not overload in the server, AC first needs to know the current available amount of resources. Second, a window control mechanism is used in the AC module to let the number of the same-type requests concurrently running on server never exceed the upper bound of window size. By keeping the maximum amount of requests of each type just below the server capacity, or the upper bound of window size, overload is prevented and peak throughput is achieved.

When AC detects the server having enough resources to serve a type-*i* request, the request scheduler is invoked to determine which classes of type-*i* requests will be sent in order to keep the proportional differentiation.

3.3.1 Offline measurement for server capacity

Because the service time of a request grows as the load of server, it is possible to determine whether the server is overloaded by watching the service time of the request. Hence, we use the service time of requests as the metric to determine whether the server is overloaded.

When the server is fully used, the service time of particular type-i requests represents the upper bound of type-i service time that the server is serving maximal number of type-i requests concurrently without overload. Thus, we can keep the service time of type-i request below the upper bound of type-i service time by adjusting the maximum number of type-i requests simultaneously running on the server.

To determine the upper bound of service time, before the web site operates practically, the service-time prober sends the probing requests to the server according to the off-line probing algorithm, as shown in Figure 3. The algorithm aims to get the upper bound of service time and the initial upper-bound of window. The main idea of this algorithm is that when the prober keeps the sending rate of requests, if the server can afford the sending rate of requests, the server will respond to the requests with the same rate. However, if the server cannot afford the requests at this sending rate, the requests will be queued in the server and the responding rate will be smaller than the sending rate.

This algorithm at first calls *WaitForServer* to lean any remaining request running on server. Next, the function *SendProbingReq* sends the probing request with the sending rate i. When we keep the sending rate i, the average number of requests concurrently running on the server is set as the initial upper-bound of window, while the average service time of requests is set as the upper bound of service time. After the results, including average responding rate, initial upper-bound of window and upper bound of service time, are collected to the ResultSet structure object *result*, the average responding rate is compared with the sending rate. Because we try to find the maximal sending rate of requests that the server can afford and can response at the same responding rate, we increase the sending rate step by step to find the maximal sending rate the server can afford. Thus,if the sending rate is equal to the responding

rate, we assume the server does not work at its full speed, and then the sending rate parameter *i* is increased by one. If the sending rate is larger than the responding rate, the algorithm will check if the server can handle the sending rate again and enters state 1. In state 1, if the server still cannot handle the sending rate, the algorithm decreases the sending rate by one and enters state 2. In state 2, if the sending rate is equal to the responding rate, we can assume that it is the maximum sending rate that the server can handle. Thus, the upper bound of service time and initial upper-bound of window can be extracted from *result*.

Note that the upper bound of service time should not be measured out when the server is overloaded, because the best throughput is not achieved when the server is overloaded [8]. The upper bound of service time in this algorithm is got at the load that the server has the best performance.

```
ResultSet OfflineProb() {
    int i=0, state=0;
    ResultSet result;
    while(true) {
        WaitForServer();                        // Clean remain requests up
        result=ProbeAndCollectResultsOnConnRate(r,i)
        If(i==result.RespondingRate) {      // Test if responding rate == sending rate
                switch(state) {
                        case 0:                     // Not reach the bound yet, increase sending rate
                                i++;
                                break;
                        case 1:                     // Not sure if i-1 is not overloaded, try again
                                state=0;
                                break;
                        case 2:                     // Find the upper bound!
                                return result;
                }
        }

        else if(i>result.RespondingRate) { // Test if responding rate < sending rate
                switch(state) {
                        case 0:
                                state=1;        // Test current sending rate i again
                                break;
                        case 1:                     // Find that both two previous tests show the server is overloaded!
                                i--;
                                state=2;        // Test the sending rate i-1 again
                                break;
                        case 2:                     // Not sure if i-1 is not overloaded, and let retry again from i-2
                                i--;
                                state=0;
                                break;
                }
        }
    }
}
```

Figure 3 Procedure of offline measurement

3.3.2 Decide the next type

To fully exhaust all types of resources, AC always selects the requests from the type that is the idlest at the server. The upper-bound of window can be used to decide which type of resources remains most. When the server is not fully utilized and can serve more than one request, AC first gets the normalized remaining window from all types of resources as

$$W_i(t) = \{\underset{i \in Types}{TotalNumReqInQueue_i(t) > 0} \mid \frac{BoundWin_i(t) - CurrWin_i(t)}{BoundWin_i(t)}\}$$

where $TotalNumReqInQueue_i(t)$ denotes the total number of type-i requests in

queue at time *t*, $BoundWin_i(t)$ denotes the window size upper bound of type-*i* requests at time *t*, and $CurrWin_i(t)$ denotes the current window size of type-*i* requests at time *t*.

Next, AC decides the idlest type of requests by selecting the maximum of them as

$$T = \arg\max_{i \in Types} W_i(t) .$$

If *T* is selected successfully, meaning that there are enough resources to serve another request, the request scheduler is invoked to decide the class of the next request.

3.4 Request scheduler

After AC decides when and which type of requests to send, the request scheduling (RS) selects a request from the class with the longest normalized system time. By this means of selection, the differentiation of normalized system time between classes is minimized and the ratios of the average system time can be approximated. The selection procedure is shown in Figure 4. At first, the algorithm calls the function *getWaitLongestClass* with the parameter *t* where *t* is the idlest resource type decided by AC. The implementations of function *getWaitLongestClass* are different, because the implementations depend on the algorithms, which are used to decide which class's request is the next to send. After the type and class is decided, the next request will be taken from queue and will be transmitted to the server. The window size which represents the number of type-*t* requests currently running on server is also added by one here.

```
void SelectClassSendReq(int t) {
    int i=getWaitLongestClass (t);
    dequeue_transmit(Queue[i][t]);
    CurrWin[t]++;
}
```

Figure 4 Procedure of type and class selection

However, the definitions of the longest normalized system time are different among varions scheduling algorithms which try to approximate PSTD. From the definition of system time in PSTD, we know that the system time is composed of queuing time and service time. Thus, we discuss how to reform three typical PDD algorithms, including WTP[3], MDP[4] and PAD[3], with the consideration of both queuing time and service time to approximate PSTD. The request scheduler uses those reformed algorithms to decide the class of the next request.

3.4.1 The reformed algorithms

The PDD algorithms are reformed to consider the queuing time and service time simultaneously. Besides, MDP and PAD are reformed to merely consider the recently departed requests, instead of overall departed requests. The change on MDP and PAD would lead to a stable differentiation in short timescale. Table 1 summarizes the properties of these reformed algorithms.

Reformed Algorithm	Metric	Suitable situation
WTP	Normalized Head-Of-Line system time	Short timescale
PWAD	Normalized averaged system time in a specified moving time window	Medium timescale
MDP	Normalized mean system time, including the system time of departed requests in a specified moving time window and the lower bound of aggregated system time of the requests currently in queue	Medium timescale

Table 1 The summary of reformed algorithms

Reforming WTP: To approximate PSTD, we reform WTP in the following. Suppose that the request scheduler decides to send the type-*j* requests, that class-*i* is

backlogged at time *t*, and that $w_{i,j}(t)$ is the queuing time of the head-of-line request in the queue of class-*i* and type-*j* at time *t*. The normalized head-of-line system time of class-*i* and type-*j* at time *t* is defined as

$$\tilde{w}_{i,j}(t) = \frac{1}{\delta_i}(w_{i,j}(t) + s_j(t))$$

where $s_j(t)$ denotes the mean service time at time *t*.

Every time a request of type-*j* is ready to sent, the WTP scheduler selects the backlogged class with the maximum normalized head-of-line system time,

$$r = \arg\max_{i \in B(t)} \tilde{w}_{i,j}(t),$$

where *B(t)* is the set of backlogged classes at time *t*.

Reforming PAD: The PAD algorithm get the mean time of a class by that of averaging all the packets departed from the class, which may exhibit a pathological behavior in short timescales as described in [3]. To avoid the behavior, we reform the PAD algorithm to average the departed requests in a fixed period *p*, and rename it to Proportional Window Average Delay (PWAD). Suppose that the request scheduler decides to send the type-*j* requests, and $d_{i,j}^m$ is the system time of the *m*th departed request of class-*i* and type-*j*, which is taken down by the response handler. Assume there was at least one departure from class-*i* and type-*j* before *t*, the normalized average system time of class-*i* and type-*j* at time *t* is defined as

$$\tilde{d}_{i,j}(t) = \frac{1}{\delta_i} \frac{\sum_{m=R_{i,j}(t)-R_{i,j,p}(t)}^{R_{i,j,p}(t)} d_{i,j}^m}{R_{i,j,p}(t)},$$

where $R_{i,j}(t)$ is the number of departed class-*i* and type-*j* requests before time *t* and $R_{i,j,p}(t)$ is the number of departed class-*i* and type-*j* requests in recently *p* seconds. Suppose that a request will be sent at time *t*. PWAD chooses the backlogged class *j*

with the maximum normalized average system time,

$$r = \arg\max_{i \in B(t)} \tilde{d}_{i,j}(t) .$$

Reforming MDP: The MDP algorithm is reformed with minor changes. Originally, MDP accumulates the system time experienced over all departed requests. We found that the properties of MDP make the system unstable after a long run, although it is actually fair. To avoid the unstable situation after a long run, we reform MDP to aggregate only departed requests in recent *p* seconds. Let $d_{i,j,p}{}^*(t)$ as the sum of the aggregate system time experienced by the requests of class-*i* and type-*j,* that have been served and taken down by response handler in recently *p* seconds, and the aggregate queuing time of the requests of class-*i* and type-*j* currently in the queue at time *t*. Let $R_{i,j,p}(t)$ be the number of type-*j* requests served from class-*i* in recently *p* seconds, $q_i(t)$ be the number of queued requests of class-*i* at time *t*. Assume no other requests of class-*i* will arrive in the future at time *t*. The normalized minimum average system time $\bar{d}_{i,j}(t)$ can be expressed as

$$\bar{d}_{i,j}(t) = \frac{1}{\delta_i} \frac{d_{i,j,p}{}^*(t) + \frac{1}{2} q_{i,j}(t)(1 + q_{i,j}(t)) \times s_j(t)}{R_{i,j,p}(t) + q_{i,j}(t)},$$

where $\frac{1}{2} q_{i,j}(t)(1 + q_{i,j}(t)) \times s_j(t)$ is the lower bound of cumulative service time experienced by all remaining class-*i* and type-*j* requests served from time *t*. Suppose that a request will be sent at time *t*. MDP chooses the backlogged class-*j* which has the maximum value of the normalized minimum average system time,

$$j = \arg\max_{i \in B(t)} \bar{d}_{i,j}(t) .$$

3.5 Response handler

The response handler aims to collect the information to keep the proportional

differentiation and the performance of the server. The handler includes three components: the content-aware response classifier, service time handler, and window adjuster. The handler update the mean service time of requests for each resource type and adjusts the window size based on the service time of the responses of probing requests.

After the gateway receives a response returned from the server, the content-aware response classifier identifies the type and class of the is response. If the response results from a probing request, the window adjuster is invoked to adjust the upper bound of the window of the probing request's type. For the other responses, the service time handler is invoked to update the average service time of the response's type.

3.5.1 Updating the mean service time

It is hard to exactly predict the service time of the requests currently in queues. However, if the service time of the same type requests is similar in short timescales, the service time averaged over recent requests of each type could be use as the predictor. Therefore, we simply average the service time of the recent 200 served type-*j* requests. Let $S_{j,k}(t)$ be the system time experienced by the *k*th served request of type-*j* at time *t*. The averaged service time of type-*j* is defined as

$$s_j(t) = \sum_{i=0}^{199} \frac{S_{j,k-i}(t)}{200}$$

3.5.2 Adjusting the upper-bound of window

The upper bound of window should be adjusted dynamically by using run time resources to avoid the waste of resources. The waste of resource results from the changing of traffic load. Because there are not always enough requests of each type to server and each type of requests uses many kinds of resources which overlap the kinds of resources required by other type of requests, when lacking one type of requests, the

server might be able to serve more requests of other types. For example, when there are no enough incoming Disk I/O-intensive requests, the server can handle more CPU-intensive requests at the same time, because Disk I/O-intensive requests also need some CPU resources.

Also, adjusting the upper-bound of window can reduce the inaccuracy of the AC. Because in our gateway, the AC is fully external to the server, the AC cannot determine the resources in the server accurately. Thus, the AC sometimes may not exhaust all resources in the server and sometimes makes the server overloaded. Adjusting the upper-bound of window lets the AC dynamic sends the requests to server according to the measured resource usages in the server. Thus, the inaccuracy of the external AC can be reduced.

To determine the ideal upper-bound of window in run time, we use the similar strategy proposed in the ATM networks. At first, the probing request for each type of requests is specified. The request should use the mean of resources that other requests of its type use. Then, the prober periodically sends the probing requests of individual types every *P* seconds. Finally, the response handler determines whether the server is overloaded by comparing the upper bound of service time and current service time of the probing request, and adjusts upper-bound of window according to the status of the server.

Notably, the incoming requests can not used to be the replacement of the probing requests, because even the incoming requests are of the same type, they may have nearly but different service time. Thus, the service time of incoming requests cannot be used as a metric to determine whether the server is overloaded.

The pseudo code of the algorithm used to adjust the upper-bound of window is shown in Figure 5. The algorithm does not adjust the upper-bound of window immediately after the response handler found that the server is not overloaded,

because adjusting upper-bound of window frequency makes the admission control become unstable. Thus, we set threshold parameters including *Imax*, which denotes the threshold of increasing the upper-bound of window, and *Dmax*, which denotes the threshold of decreasing the upper-bound of window, to prevent the upper-bound of window from variation. Every time when the service time of a type-*j* request is greater than the upper bound of type-*j* service time, the parameter *adjCounter* is added by one. Otherwise, the *adjCounter* is subtracted by one. If *adjCounter* exceeds the parameter *Dmax*, the upper bound of type-*j* window size is subtracted one and *adjCounter* is set to zero. If *adjCounter* is lower than the parameter *Imax*, the upper bound of type-*j* window size is added one and *adjCounter* is set to zero.

Based on the experiment results, we found that *Imax* -9 and *Dmax* 4 are suitable to reduce the variation of the upper-bound window.

```
int AdjustWindow(response r) {
     int t;
     t= getType(r);
     if( r.servicetime>=bound_time[t] ) adjCounter++;
     else adjCounter--;
     if(adjCounter>Dmax && bound_window[t] >1) {
          bound_window[t]--;
          adjCounter=0;
          }
     else if(adjCounter<Imax) {
          bound_window[t]++;
          adjCounter=0;
          }
}
```

Figure 5 Procedure of dynamic window size adjusting

Chapter 4 Experiment and Results

In this Chapter, we first implement MR-PSTD in Squid, which is an open source package of web proxy. Next, the experiment environment and network topology is introduced. The experiment results shows that the differentiation effects for WTP and MDP meet the pre-specified differentiation ratios and performance improvement is significant using the MR-PSTD web gateway.

4.1 Software Implementation

The implementation of MR-PSTD gateway is based on the Squid package. Figure 6 illustrates the processing flow in the modified Squid. We insert our MR-PSTD components in the function that Squid starts to forward requests and the function that Squid finishes to receive responses. Table 2 shows the new added functions in the processing flow of Squid. The structure *request_t* keeps the status of clients needed by Squid, and the structure *FwdState* keeps the extra status after Squid forwards requests to server.

If the original Squid receives a request whose response is not in cache, it immediately forwards a request to the server. Because the MR-PSTD gateway need to queue the requests for rescheduling and admission control, after Squid receives whole requests, we change the processing flow of the function *fwdStart()*. The modified *fwdStart()* does not forward the request immediately, but it invokes *reqClassifier()*, which is our content-aware request classifier implementation. The function *reqEnqueue()* is invoked next, and all necessary information for future processing is stored in queue.

The admission window control is implemented in the function *selectResType()*, which implements the algorithm in Section 3.3.2. When the function *selectResType()*

decides which type of requests is suitable to send, the function *selectClassInType()*, which is the request scheduler and implements one of the algorithms in Section 3.4, decides which class of requests is the most suitable to send. Because we implement three different algorithms including WTP, PWAD and MDP to approximate the PSTD model, the implementations of the function *selectClassInType()* are different among algorithms. The functions *reqDequeue()* and *reqTransmit()* are invoked next to send the request to the server.

After the sever sends the response to Squid, the function *httpReadReply()* receives the whole response and invokes *fwdCompete()* next. We modify the function *fwdCompete()* and implement the Response Handler in it. Thus, we insert our content-aware response classifier function *resClassifier()*, the service time averaging implementation *updateData()*, and on-line window upper bound adjusting implementation *adjustWindow()* at the last of the function *fwdCompete()*. The function *storeCompete()*, which starts to send the response to the client, is invoked after all newly added functions.

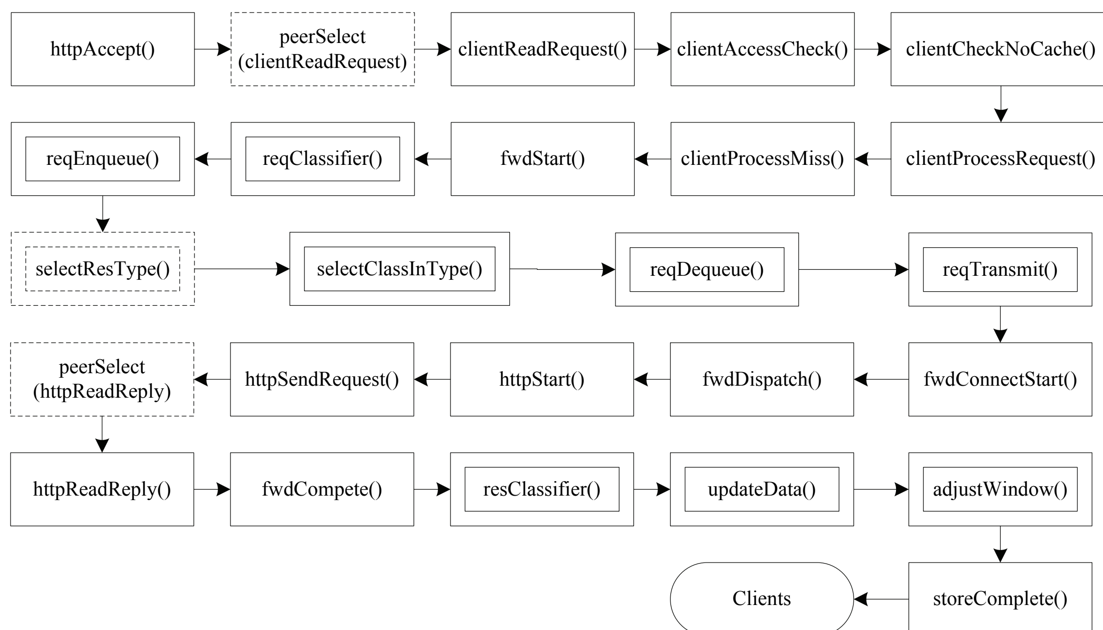


Figure.6 The modified processing flow of Squid

Function Name	Function Description
resClassifier(request_t*)	Content-aware classifier, which identify the type and class of the request
resEnqueue(FwdState*, request_t*, int, int)	Put the received request into the queue by its type and class.
selectResType()	Decide which type of requests to send.
selectClassInType(int)	Decide which class of requests to send.
reqDequeue(int, int)	Take the request from the queue.
reqTransmit(FwdState*, request_t*)	Send the request to the server
resClassifier(FwdState*)	Content-aware classifier, which identify the type and class of the response
updateData()	Update the service time corresponding to the type of the response
adjustWindow()	Adjust window upper bound

Table 2 The description of new functions

4.2 Test Bed

Figure 7 is the topology of the experiment, which composes of a client PC, two server PCs, a MR-PSTD gateway PC, and two 100Mbps Ethernet switches. Each PC has a 2.8GHz Pentium 4 CPU, 512 MB RAM, a 200G 7200 RPM hard disk, and a 100 Base-T Ethernet interface. The two-tier web site consists of.the two server PCs. One server machine runs the web server, application server software, and file access server software written in PHP, while the other contains the database server software. The client machine drives the web site with a workload generator written in C, and the workload generator can emulate 225 virtual users at once. Table 3 shows the software running on those PCs.

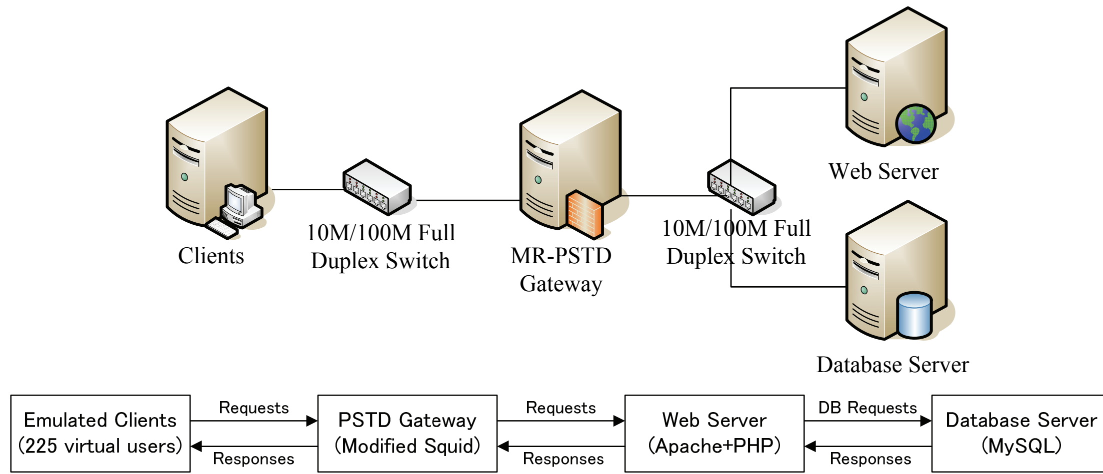


Figure.7 Experiment network topology

Purpose	Software
Operation system	Linux 2.6.12 (Debian)
Web server	Apache 2.0.55
Application interpreter	PHP 4.4.2
Database server	MySQL 5.0.18

Table 3 Software running on PCs

Before the experiments starts, we have to classify requests into classes and types, and set parameters, including parameter p in PAWD, parameter p in MDP, initial window size of each type of requests, and upper bound service time of requests of each type. In our experiments, we divide the requests into three classes C1, C2, and C3. In each class, there are three types of requests, including CPU-intensive requests, Disk I/O-intensive requests, and Database-intensive requests. Thus, there are nine individual groups of requests. We also define the system-time differentiation ratios to 1:2:4, and the parameters p in PWAD and MDP are both set to 120 seconds.

To have the initial window size and upper bound service time of requests of each type, before the MR-PSTD gateway starts to operation, we have to run the offline probing procedure shown in Figure 3. After the offline probing procedure, we have the upper bound service time and initial window size as listed in Table 4.

Type of request	Initial window size	Upper bound service time (sec)
CPU-intensive requests	2	0.45
Disk I/O-intensive requests	7	0.42
Database-intensive requests	5	0.38

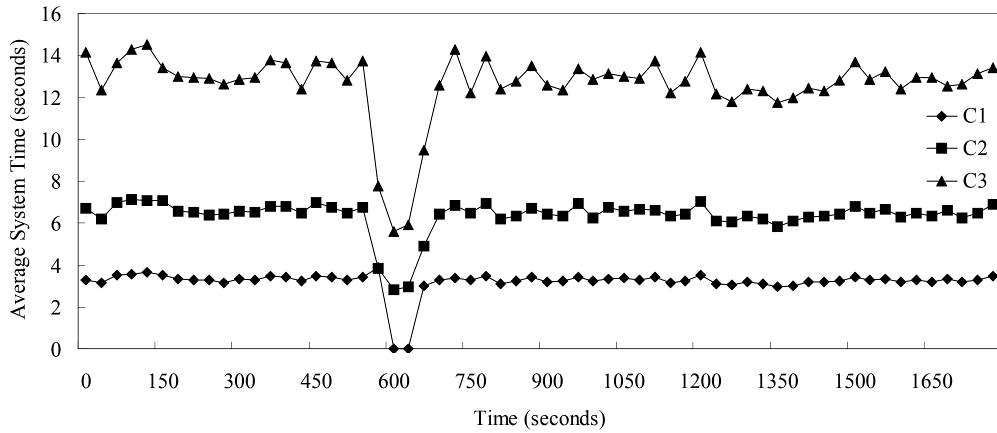
Table 4 Experiment results of offline probing

4.3 Effects on Differentiation

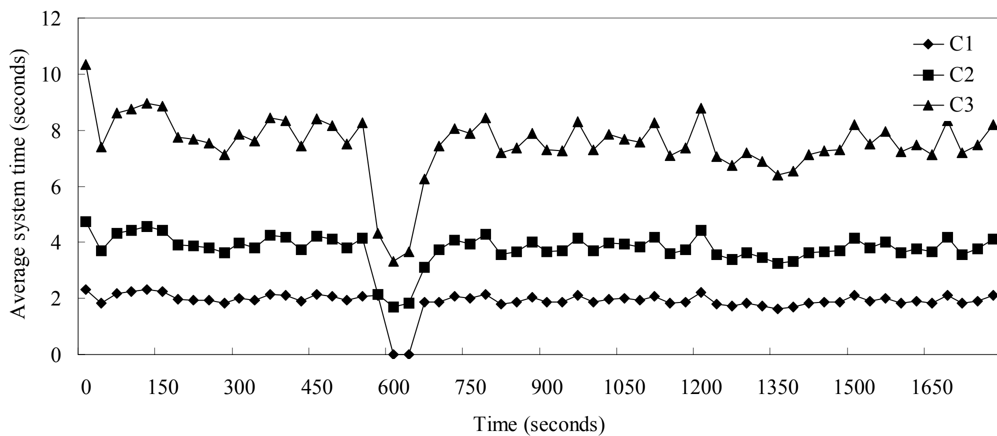
In the following trial, the workload generator always keeps 75 virtual users for each class, which are composed of 25 virtual users for three types of requests in the class. Because the empty server at the initial stage serves requests much faster then the fully loaded server, the window adjusting mechanism cannot determine correctly at the initial stage. Hence, we let the window adjusting mechanism waits for a minute after the trial starts. The virtual users of class 1 will stop sending requests at the 570th second for 90 seconds to test if the algorithm can tolerate rapid change of traffic while keeping the proportional differentiation between classes.

4.3.1 WTP is stable in short timescales

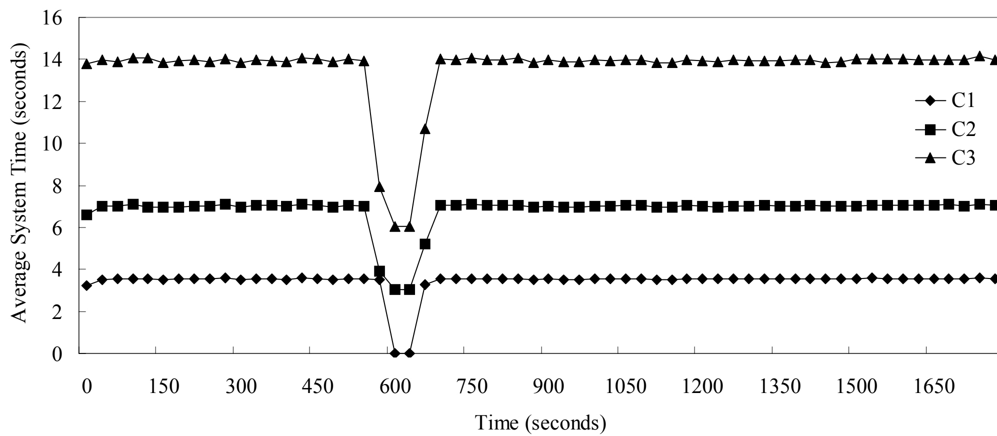
Figure 7(a) shows that WTP is particularly suitable to apply on the MR-PSTD gateway because WTP can converge to the stable state in a few seconds even after the rapid change of traffic. After the WTP-based MR-PSTD gateway runs for 30 minutes, the averaged system time differentiation ratio in last 10 minutes between classes is 1:1.92:3.81, and it is very close to the pre-specified ratio.



(a) Average system time among classes



(b) Average system time of Disk I/O-intensive requests



(c) The average system times of database-intensive requests

Figure.7 Averaged system time of WTP-based MR-PSTD gateway

The experiment results shown in Figure 7(b) reveal that the system time of Disk I/O-intensive requests is the most unpredictable. Because the operation system, Linux

2.6.12, tends to accumulate the Disk I/O-intensive requests and serves the entire backlog requests at the same time, the service time of Disk I/O-intensive requests are not stable. Thus, it also results in the vibration in Figure 7(a).

In Figure 7(c), the system time of Database-intensive requests, which is much stable than that of Disk I/O-intensive requests, shows that the WTP algorithm is not influenced by the rapid change traffic because of WTP's properties. Although Disk I/O-intensive requests cause the vibration of average system time of total requests in short timescale, the differentiation ratio between classes is kept in acceptable range.

4.3.2 PWAD is unsuitable for PSTD

Figure 8 shows that PWAD is unsuitable to be employed in the MR-PSTD gateway, because it is extremely unstable. After the PWAD-based MR-PSTD gateway runs for 30 minutes, the averaged system time differentiation ratio in the last 10 minutes between classes is 1:1.63:3.23, and it is far from the pre-specified ratio. The unstable results and pathological behavior comes from its basic properties because PWAD only considers departed requests but ignores future requests. When the system starts to work and warms up, because the server is empty at start time, the server serves requests fast, and C2 and C3 requests have short normalized system time, which is much smaller than the normalized system time of C1 requests, before the server becomes busy. Thus, the normalized average system time of departed C1 requests is longer than those of departed C2 and C3 requests, which causes PWAD sends only C1 requests to reduce the normalized average system time of C1. After the departed requests of C2 and C3 are discarded by the moving window averaging mechanism, PWAD turns to send C2 and C3 requests since they are already queued for a long time. Thus, the normalized averaged system time of C2 and C3 becomes very large and PWAD do not send C1 requests anymore until the departed requests of C2 and C3 experiencing long queuing time are discarded. Besides, when the

algorithm decides to send only C2 and C3 requests, C1 requests experience long queuing time.

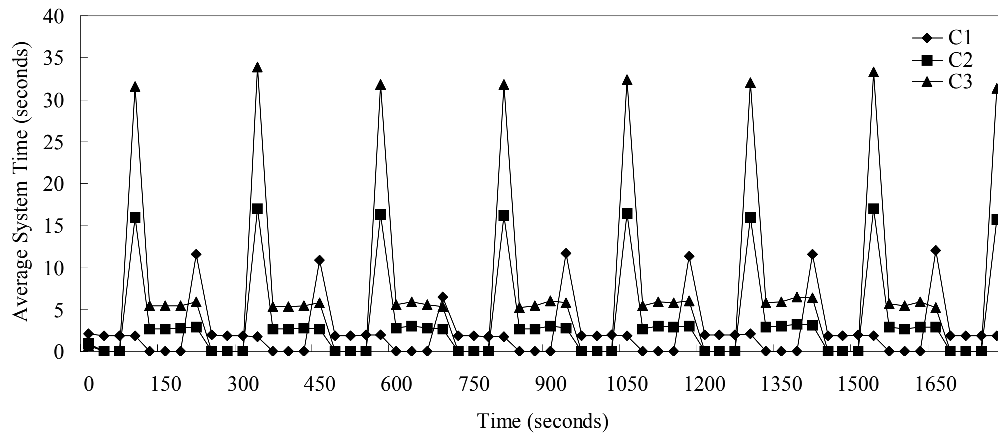
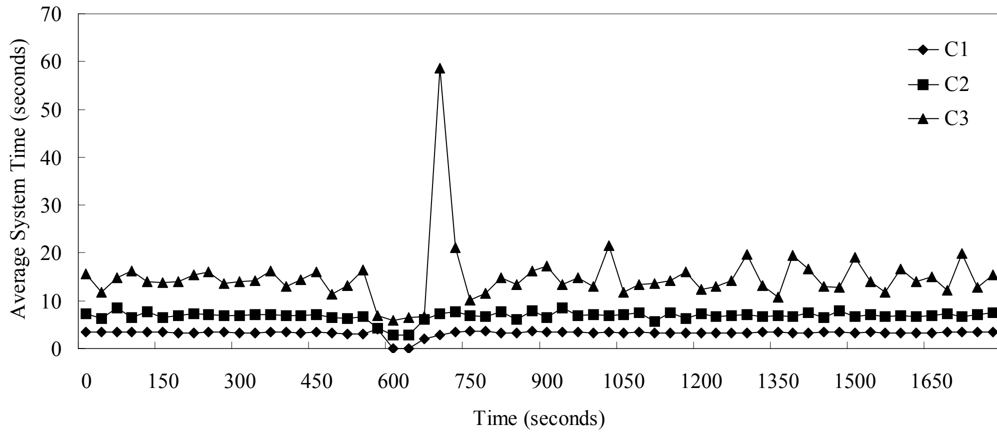


Figure 8 Averaged system time of PWAD based MR-PSTD gateway

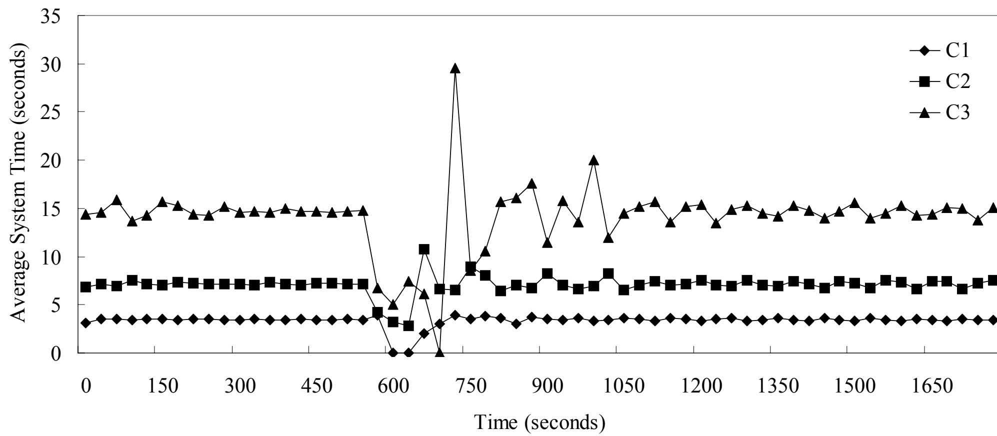
4.3.3 MDP is stable in medium timescales

Figure 9 shows that MDP is suitable to be employed in the MR-PSTD gateway, because the reformed MDP not only keeps only w seconds information of departed requests but also considers the lower bound system time of requests currently in queue. Although MDP never causes pathological variation of system time as PWAD in long timescale, after a rapid change of traffic, MDP gets unstable for about 100 seconds. The wavelet of system time, which results from redeeming the gap of normalized mean system time, costs about 400 seconds to converge. However, in another point of view, MDP can keep the differentiation ratio in medium timescale while WTP cannot keep it, because MDP redeems the normalized averaged system time in recent p seconds, which is defined to 120 seconds in our experiments.

After the MDP based MR-PSTD gateway starts for 30 minutes, the averaged system time differentiation ratio in last 10 minutes between classes is 1:2.03:4.19, and it is very close to the pre-specified ratio.



(a) Averaged system time among classes



(b) Averaged system time of database intensive requests

Figure 9 Averaged system time of MDP based MR-PSTD gateway

In Figure 9(b), the result of database intensive requests shows that the MDP algorithm wants to redeem the gap, when C1 traffic is suspended. The C3 and C2 traffic experienced higher system time right after the C1 traffic restore. However, after about 400 seconds, the wavelet of the system time of C3 finally gets smooth.

The results show that MDP is suitable for the web site that has stable traffic and need more accuracy on the effect of service differentiation, because MDP always keeps the high priority classes have better service quality which is proportion to its differentiation ratio in the specified timescale.

Comparing with MDP, WTP does not always keep the differentiation ratio in

medium timescale because WTP does not redeem the gap after rapid change of traffic, but WTP is always stable in short timescale. Thus, in the environment that the traffic changes frequently, WTP may be more suitable than MDP. Among all of the three algorithms, WTP is the most suitable algorithm to approximate PSTD model in the environment that the traffic changes frequently, and MDP is the most suitable algorithm in the environment that the traffic in stable.

4.4 Performance Improvement

Beside MR-PSTD, we implement a single-resource PSTD (SR-PSTD) web gateway and a PSTD web gateway without any resource control mechanism, in order to demonstrate the effectiveness of the multiple-resources window control mechanism.

The PSTD web gateway does not consider resource issues. The requests are only classified into classes and put into individual queues. The PSTD web gateway always tries to forward requests once they arrive in queues. In the SR-PSTD web gateway, the requests are also classified into classes and put to individual queues. Different from PSTD gateway, SR-PSTD gateway tries to handle the resource issues and considers that the server has only one kind of resources. Thus, it uses only one window control mechanism. In the SR-PSTD gateway, the upper-bound of service time is set to 0.55 seconds, which is measured by our offline probing procedure.

After experiments using WTP, MDP and PWAD as the algorithm to approximate the PSTD model, we found that the difference of algorithms is not effect the throughput of the server behind the gateway. Thus, we only present the results over 30 minutes, which is aggregated from the gateway using WTP algorithm to approximate the PSTD model. In Figure 10, the throughput of MR-PSTD is 1.78 and 1.58 times of that of PSTD and SR-PSTD, respectively, while the system time of MR-PSTD, as

shown in Figure 12, is the smallest, because the overloading in PSTD gateway and non-well utilized resources in SR-PSTD gateway, The significant result shows that multiple resources management is necessary and important in server-side QoS.

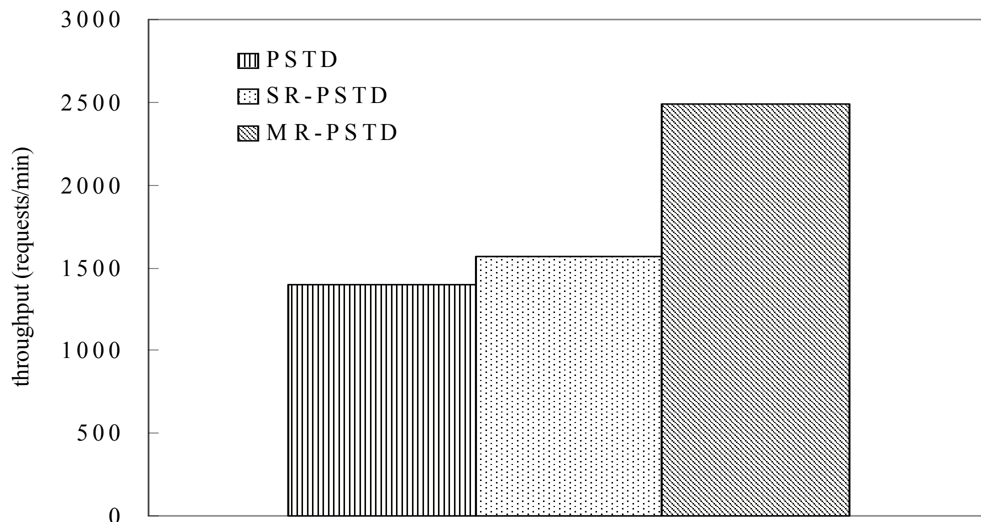


Figure 10 Average throughputs between gateways

Because the throughput of SR-PSTD gateway is far from MR-PSTD gateway, to avoid any fault resulted from incorrect parameter setting, we try to adjust the upper bound of service time manually to verify that the result of the server behind the SR-PSTD gateway shown in Figure 10 is correct. In Figure 11, the best throughput of the server behind SR-PSTD gateway is achieved when the service time upper bound set to 0.55 seconds, which is equal to the upper bound of service time measured by our offline probing procedure. Thus, we can confirm that the result shown in Figure 10 is correct because the server behind the SR-PSTD gateway already runs on its full speed, and our offline probing procedure is effective to measure the upper bound of service time.

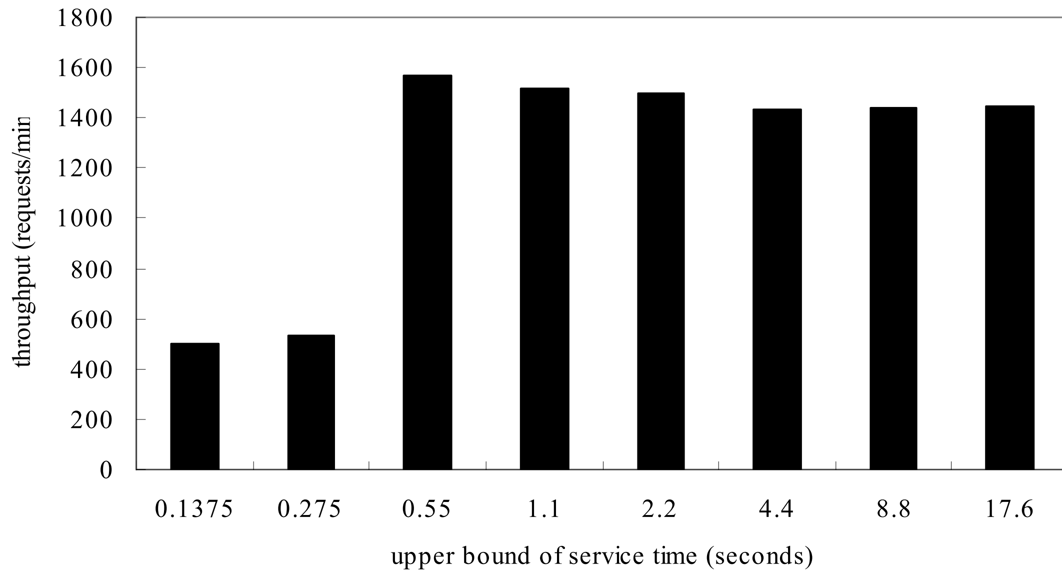


Figure 11 Throughput with different upper bound of service time

In Figure 12, the mean queuing time of requests in the SR-PSTD gateway is much longer than that in the MR-PSTD gateway. Because the SR-PSTD gateway can not estimate the resource usages accuracy, the SR-PSTD gateway sends too few requests to exhaust all resources at the server, causing that the throughput is degraded. The service time of PSTD is much longer than that of SR-PSTD and MR-PSTD because the PSTD-based server is already overloaded and the server spends much more time to handle every request.

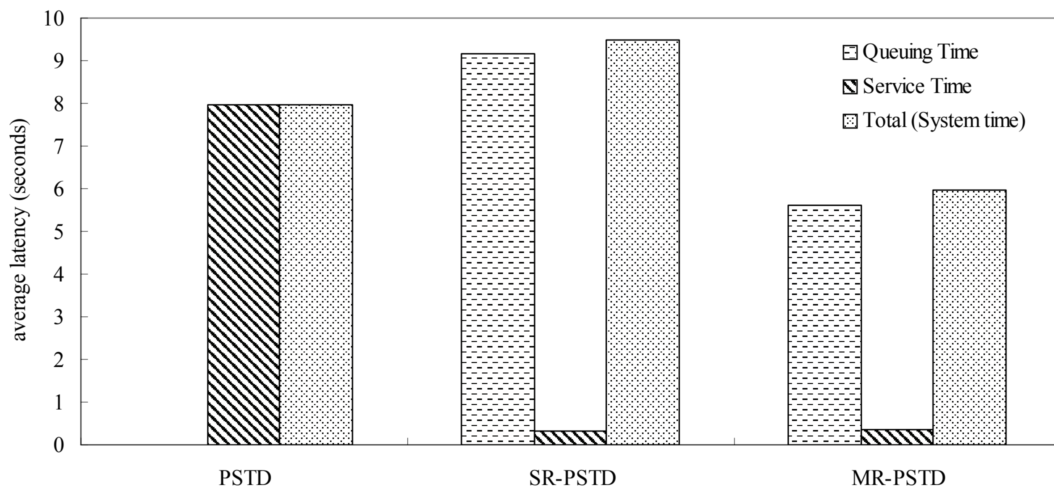


Figure 12 Service time and queuing time distribution between gateways

In Figure 13, the total number of requests concurrently running on the server behind the PSTD gateway is much larger than those behind the SR-PSTD and MR-PSTD gateway. This is results from that the PSTD gateway sends requests to the server unlimitedly because the PSTD gateway does not contain any admission control module. Also, because of the overhead of frequent content switching between requests in the server, the throughput of PSTD-based server is low as shown in Figure 10. From the total number of requests concurrently running on the server behind SR-PSTD and MR-PSTD gateway, we can find that some resources in the server behind SR-PSTD gateway may be still idle, because the number of total requests concurrently running on the server behind SR-PSTD gateway is less than the server behind MR-PSTD gateway.

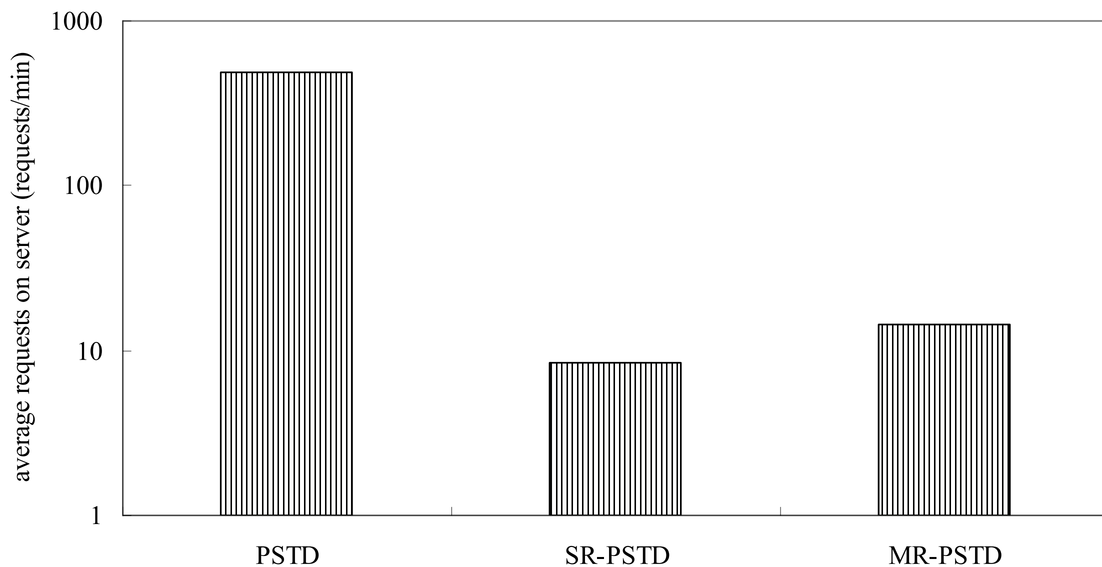


Figure 13 Number of concurrent requests between gateways

Chapter 5 Conclusion and Future Works

This work first proposes a Multiple-resources Proportional System-Time Differentiation (MR-PSTD) model. It is more desirable than PDD and PSD for server QoS, because MR-PSTD not only provides proportional system-time differentiation for users, but also improves the throughput of the server by optimizing the usage of resources in the server.

Next, a gateway system is proposed to approximate the PSTD model with multiple-resources consideration. The system is easy to be deployed because it is external to the website and does not need to modify the website solution. We reform three PDD algorithms, WTP, PWAD and MDP to consider the service time in scheduling request. Besides, we design an admission control module to prevent the servers from overload while exhausting all types of resources.

To prove the effectiveness of our gateway, we implement it by modifying Squid, and put it between clients and web servers. The experiment results show that the MR-PSTD gateway increases the peak throughput by 78% and decreases the system time by 25%, comparing with the original website system. This is resulted from that all types of resources at the server are exhausted. The results also show that both WTP and MDP are suitable to be employed in the MR-PSTD gateway because they achieve the pre-specified differentiation ratio, while PWAD is not suitable at all.

In the future, we plan to improve the mean system-time calculating scheme, because the present gateway cannot accurately predict the service time of the Disk I/O-intensive requests due to their large variation service time. In addition, it is possible to further simplify the present gateway system by predicting the distribution of requests based on the queuing theory.

References

[1] X Zhou, J Wei, and C Xu, "Processing Rate Allocation for Proportional Slowdown Differentiation on Internet Servers," *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* Apr. 2004

[2] J Wei, X Zhou, and C Xu, "Robust Processing Rate Allocation for Proportional Slowdown Differentiation on Internet Servers," *IEEE Transactions on Computers, vol. 54, Issue 8*, Aug. 2005

[3] C Dovrolis, D Stiliadis, and P Ramanathan, "Proportional Differentiated Services: Delay Differentiation and Packet Scheduling," *IEEE/ACM Transactions on Networking, vol. 10, Issue 1*, Feb. 2002

[4] T Nandagopal, N Venkitaraman, R Sivakumar, and V Barghavan, "Delay Differentiation and Adaptation in Core Stateless Networks," *Proceedings of IEEE INFOCOM 2000, Tel-Aviv, Israel*, Apr. 2000

[5] H Shimonishi, I Maki, T Murase, and M Murata, " Dynamic Fair Bandwidth Allocation for DiffServ Classes", *IEEE International Conference on Communications, 2002. ICC 2002, vol. 4*, Apr. 2002

[6] EC Park, and CH Choi, "Proportional Bandwidth Allocation in DiffServ Networks," *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, vol. 3*, 2004

[7] SC Lee, JC Lui, and DK Yau, "A Proportional-Delay Diffserv-Enabled Web Server: Admission Control and Dynamic Adaption," *IEEE Trans. Parallel and Distributed Systems, vol. 15*, May. 2004

[8] HU Heiss and R Wagner, "Adaptive Load Control in Transaction Processing Systems," *In 17th International Conference on Very Large Data Bases, Barcelona, Spain*, Sep. 1991