Implementation of a New Scheduling Policy in Web Servers

A Thesis
Submitted to
The Faculty of Information Technology

by

Ahmad S. Al-Sa'deh

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

Supervisor
Prof. Adnan H. Yahya

Graduate Program in Scientific Computing
Birzeit University
July 2007

Implementation of a New Scheduling Policy in Web Servers

Approved by:

Prof. Adnan Yahya, Advisor
Computer System Engineering Department,
Birzeit University, Palestine.

Dr. Hussein Badr
Associate Professor,
Computer Science Department,
Stony Brook University, USA.

Dr.Bassem Sayrafi
Assistant Professor,
Computer Science,
Birzeit University, Palestine.

Date Approved _____

To my parents

To my wife

To my brothers and sisters

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor Professor Adnan Yahya for his inspiration and encouragement during my graduate education. He was very patient and always provided his excellent guidance in helping me to complete this thesis. I would like to convey my sincere thanks to Dr. Hussein Badr for the time he spent sharing his vast wealth of knowledge with me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACK	: Acknowledgement
ADSL	: Asymmetric Digital Subscriber Loop
bps	: Bits Per Second
cbq	: Class Based Queue
CPU	: Central Processing Unit
CT	: Current Time
cwnd	: Congestion Window
Diffserv	: Differentiated Service
ET	: Elapsed Time
FCF	: Fastest Connection First
FIFO	: First In, First Out
FS	: File Size
FSP	: Fair Sojourn Protocol
HTTP	: Hypertext Transfer Protocol
IOBUFSIZE	: Input Output Buffer Size
IP	: Internet Protocol
LAN	: Local Area Network
MB	: Megabyte
MSS	: Maximum Segment Size
netem	: Network Emulation
NIC	: Network Interface Card
NIST	: National Institute of Standard and Technology
pfifo_fast	: Packet First-In, First-Out
prio	: Priority
PS	: Processor Sharing
qdisc	: Queuing Disciplines
QoS	: Quality of Service

RAM : Random Access Memory

RFS : Remaining File Size

RRT : Remaining Response Time

RST : Request Start Time

RT : Response time

RTT : Round-Trip Time

SEDA : Staged Event-Driven Architecture

sfq : Stochastic Fair Queuing

SPEC : System Performance Evaluation Cooperative

SRPT : Shortest Remaining Processing Time

SRRT : Shortest Remaining Response Time

SSL : Secure Socket Layer

ssthresh : Slow Start Threshold

STERT : Shortest Total Estimated Response Time

SURGE : Scalable URL Request Generator

SYN : Synchronize sequence numbers

tbf : Token Bucket Flow

tc : Traffic Control

TCP : Transmission Control Protocol

TERT : Total Estimated Response Time

TOS : Type of Service

UE : User Equivalents

URL : Uniform Resource Locator

WAN : Wide Area Network

WWW : World Wide Web

ABSTRACT

A widely encountered problem in web servers over the Internet is the long response time. It is possible to reduce the response time of requests at a web server by simply changing the order in which we schedule the requests. Recently, the Shortest-Remaining-Processing-Time (SRPT) has been proposed for scheduling requests in web servers. The SRPT assumes that the response time of the requested file is strongly proportional to its size. However, depending only on the size of the file for determining the priority of the request is not enough, since it doesn't take into consideration the client-server interaction through the Internet, where web servers are mainly used. In the Internet, the clients are geographically dispersed which presents high diversity in path bandwidth, round-trip time and packet loss characteristics. To account for these parameters, this thesis proposes a new scheduling policy for processing static HTTP requests in web servers that better estimates the response time. We call this policy, Shortest-Remaining-Response-Time (SRRT).

Our approach benefits from the TCP implementation to capture useful scheduling information about the interaction between the server and the client through the network. The SRRT prioritizes requests based on a combination of the current Round-Trip-Time (RTT), TCP window size and the size of what remains of the requested file. The requests which have the shortest estimated remaining response time receive higher priorities.

The implementation is done at the kernel level for controlling the order in which socket buffers are drained into the network. Our experiment uses the Linux operating system and the Apache web server. In the experiment the requests are generated by the Scalable URL Request Generator (SURGE) workload generator, and the WAN is represented by Network Emulation (netem).

We compare SRRT to SRPT and processor-sharing (PS) policies. SRRT and SRPT show an improvement over PS. However, the SRRT shows the best improvement in the

mean response time. SRRT gives an average improvement of about 7.5% over SRPT for both 10Mbps and 100Mbps links and under all loads. For 10Mbps link, the maximum improvement of SRRT over SRPT is 13.2%. While for the 100Mbps link the maximum improvement is 11.6%.

This improvement comes at a negligible expense in response time for long requests. We found that under 100Mbps link, only 1.5% of long requests have longer response times. The longest request under SRRT has an increase in response time by a factor 1.7 over PS. For 10Mbps link, only 2.4% of requests are penalized, and SRRT increases the longest request time by a factor 2.2 over PS.

CHAPTER I

INTRODUCTION

In the past few years the World Wide Web (WWW) has become extremely popular; not only for communication and browsing but also for conducting business and selling on the Internet. Internet traffic continues to increase rapidly, having almost doubled every year since 1997 [1]. Also the number of Internet users increases by the day. The growth in the Internet users increases at an annual rate of 18% and is estimated at one billion users in 2005 [2]. A second billion users will join in the next ten years [2]. In June 2007, the Internet usage reaches to 1,133,408,294 users [3]. A popular web site like *google.com* received 91 millions web searches per day in the United States in March 2006 [4].

As the demand on the Internet/web servers grows; the web servers performance becomes important objects of study and evaluation. Today busy web servers are required to service many clients simultaneously, sometimes up to tens of thousands of concurrent clients [7]. If a busy web server's total request rate increases above the total link capacity or the total server capacity (maximum simultaneous/concurrent users), the number of rejected requests increase dramatically and the server offers poor performance and long *response time*, where the response time of a client is defined as the duration from when the client makes a request until the entire file is received by the client.

No one likes to wait and the clients will get frustrated if they cannot complete their requests within a certain time. The slow response times and difficult navigation are the most common complaints of Internet users [5]. Research shows the need for fast response time. The response time should be around 8 seconds as the limit of people's ability to keep their attention focus while waiting [6]. The question arises, what can we do to improve the response time?

1.1 Statement of the problem

This study considers how we might improve the response time for clients accessing a busy web server by applying the scheduling policies on web servers. A web server cannot handle numerous requests at the same time, and will typically use a buffer or queue to store incoming requests awaiting service. Requests in queue are typically stored in order of arrival. The web server will take the request at the front of the queue, and serve it first. This is an example of first-in-first-out (FIFO) scheduling. A more common scheduling policy in web servers is *processor sharing* (PS) scheduling. In PS each of n competing requests (processes) gets $1/n$ of the CPU time, and is given an equal share of the bottleneck link. This approach is fair, and prevents long flows from monopolizing server resources. It has been known from the queuing theory that Shortest Remaining Processing Time (SRPT) scheduling policy is an optimal algorithm for minimizing mean response time [18]. However, the optimal efficiency of SRPT depends on knowing the response time of the requests in advance. M. Harchol-Balter et al.[21], [22] used the job size to refer to processing time (response time) of the job to implement SRPT for web servers to improve user-preceived performance. In the Internet environment, depending only on the file size for estimating the response time is not enough since it does not take into consideration the Internet parameters like Round Trip Time (RTT), bandwidth diversity, and loss rate. To include these parameters in estimating the user response time we proposed a new scheduling policy in web server which is called Shortest Remaining Response Time (SRRT) to improve the mean response time of clients.

1.2 Problems with existing algorithms

Recently, size-based scheduling policies on web servers [8], [9], [11], [21] assumed that the response time is only the size of the file being served, which is well-known to the server. And since the small files were common in the web, the performance improvements were possible by improving the server's services for small files [14]. SRPT algorithm reduces the mean response time without hardly penalizing the large requests [21]. However, is file size alone a good estimator to clients' response time from the web servers?

Web servers are used mainly in the Internet to service requests from a large number of

heterogeneous users. The diversity of the user characteristics in their connectivity conditions, presents high variability in bandwidth, round-trip times and packet loss. Therefore, the size-based scheduling policies for web servers should take into account the Wide Area Network conditions (WAN) in their implementation in addition to file size as a main criterion for the web server scheduling.

Other work on web server scheduling [16], [25] show that in the Internet environment we cannot ignore the effect of the WAN parameters like network path delay and packet loss in the scheduling algorithms on web servers. Dong Lu et al. [25], have shown that the correlation between the file size and the response time are quite low, and shows that the performance of size-based scheduling on web servers degrade dramatically due to weak correlation between the file size and the response time. In Fastest Connection First (FCF) [16] giving the priority to HTTP request based on the request size and on the throughput of the user's connection improves the mean response time. FCF work is done only by simulation. Mayank Rawat et al. [10] proposed the SWIFT scheduling algorithm for web servers. They prioritize the request based on both the size of the requested file and the round trip time (RTT). The SWIFT algorithm shows a better improvement over the SRPT scheduling algorithm. In the SWIFT algorithm implementation, they assumed that the HTTP requests are embedded with the RTT in their trace driven experiment. This assumption is not a realistic scenario for measuring the RTT "on-the-fly". Our approach, Shortest Remaining Response Time First (SRRT), uses *getsockopt()* Linux system call to get the RTT value and the TCP window size on fly for each connection. The RTT and the TCP window size determine the maximum throughput of the TCP connection. Thus, the RTT and the TCP window size were considered as new parameters for estimating the response time in addition to the size of the requested file.

1.3 Response time components

The client-server interaction over the Internet is shown in Figure 1. According to this model, the total response time can be defined as the time from the client submits a request until the time he receives the last byte of the requested file which can be classified by the

following delay categories:

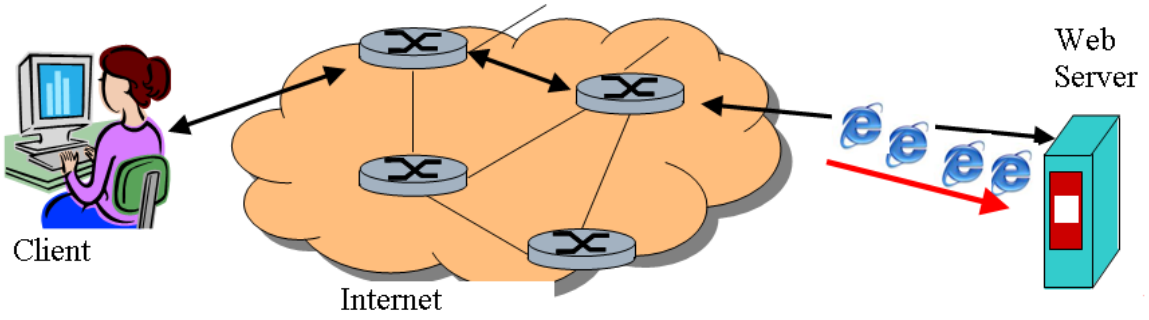


Figure 1: Client-server interaction over the Internet

1. *Queuing delay*: The time a packet (request/replay) waits in a queue at the client/server until it can be transmitted. This delay depends on the load on the communication link. Also, queuing delay is proportional to the queue (buffer) size. The average waiting time in the queue increases as the number of waiting packets to be transmitted in the queue increases. However, this is preferable to a shorter queue capacities which would lead to drop packets, which is a cause of much longer overall transmission times due to retransmission.

2. *Transmit delay*: The delay between the transmission of first bit of the packet (request/replay) to the transmission of the last bit. This delay depends on the capacity of the communication link.

$$Transmit\ Delay = \frac{Transfer\ Size}{Link\ \ Capacity} = \frac{S}{C}$$

   We shall consider the transmission delay of a request to be negligible, given the small size of request packets.

3. *Server processing delay*: The time from receiving a request/packet at the server to the point of putting the replay into the server transmit queue. This time delay depends on the CPU speed and CPU load in the server.

4. *Round Trip Time Delay (RTT)*: The total time for the request to propagate through the network to the server, and the replay to propagate back to the client. This delay

depends on the network speed, the distance between the transmitter and the receiver, and the congestion on the network.

So the total response time (*RT)* is:

$$RT \approx RTT + \frac{S}{C} + \textit{Queue Delays}$$

The mean response time ($\overline{RT}$) is the main performance metric in this study which is defined as the average response time for all n requests:

$$\overline{RT} = \frac{\sum_{r=1}^{n} RT(r)}{n}, \; r \text{ is the request.}$$

1.4 Our approach

We propose an effective method for estimating the response time for a web client using the TCP implementation at the server side only, without introducing extra traffic into the network or even storing historical data on the server. The proposed method finds the client response time in each visit to a server, and then schedules the requests based on the Shortest Remaining Response Time First (SRRT). In this method we try to have the exact knowledge of the response time by taking into consideration the WAN conditions such as the path delay and bandwidth, the traffic on the network and the congestion, the load of the server, and the size of the file.

In our approach we use the Linux *getsockopt()* system call to get the RTT and the congestion window (*cwnd*) from the TCP implementation at the server side. These two values are used to estimate the maximum TCP transmission rate for the connection between the client and the server. The response time for each visit to a server can be approximated by the RTT plus the size of the requested file divided by the estimated TCP transmission rate. where the TCP transmission rate is defined by the congestion window size (*cwnd*) divided by the RTT value. Requests are then prioritized based on the shortest remaining response time request first. See section 3.4.5 for the complete description of SRRT algorithm.

For our implementation, we use a web workload generator to generate requests with certain distribution and focus only on static requests which form a major percentage of the web traffic [15], [21]. Our experiment uses the Linux operating system and Apache web

server . The experiment setup is detailed in chapter 4.

We compare the SRRT with the PS and SRPT scheduling policies in web servers. We find that the SRRT gives the minimum mean response time. We conclude that the client response time is affected by the Internet traffic. So any scheduling policy in web servers should take into consideration the Internet conditions to prioritize the requests.

1.5 Organization of the thesis

The thesis is organized into six chapters. Chapter 2 discusses relevant previous work in web server requests scheduling. The scheduling policy design considerations that we made and implementation details are covered in chapter 3. Chapter 4 describes experiment setup and configuration details. The experiment results and analysis are given in chapter 5. Finally, chapter 6, summarizes the research performed and discusses possible future work.

CHAPTER II

LITERATURE REVIEW

There have been numerous studies about assigning different priorities to connections for trying to improve the web server performance; especially the client response time. The work can be divided into three categories; web server request scheduling, admission control, and service differentiation.

2.1 Web server request scheduling

Request scheduling refers to how requests are stored and served by the server. Usually, the operating system is responsible for request scheduling. It is well known from scheduling theory literature [17], [18], [20] that if the task sizes are known, the Shortest Remaining Processing Time first (SRPT) scheduling is optimal for reducing the queuing time, therefore reducing the mean response time. Here, we will focus on scheduling algorithms that are based on SRPT and how one can apply them to the web servers.

There are many techniques based on the SRPT algorithm. These techniques for web server scheduling can be divided into three categories: web server scheduling implementation, web server scheduling simulation studies, and web server scheduling analytical/theoretical studies.

2.1.1 Web sever scheduling implementation:

The work that implements scheduling for web servers based on the SRPT was done on both the application level, and at the kernel level to prioritize HTTP requests at the web server. M. Crovella et al. [8] experimented with the SRPT connection scheduling at the application level. They get an improvement in the mean response times, but at the cost of drop in the throughput by a factor of almost 2. The problem was that they did all the modifications to the Apache web server at the application level. As a result, there is no adequate control over the order in which the operating system services the requests.

M. Harchol-Balter et al. [21] implemented SRPT connection scheduling at the kernel level. They get much larger performance improvements than in [8] and the drop in the throughput was eliminated. B. Schroeder and M. Harchol-Balter [22] show an additional benefit from performing SRPT scheduling for static content web requests. They show that SRPT scheduling can be used to alleviate the response time effects of transient overload conditions without excessively penalizing large requests.

SWIFT: Scheduling in Web Servers for Fast Response time algorithm [10] extend the work in [21] based on SRPT, but taking into account in addition the size of the file, the round-tip-time (RTT) to represent the distance between the client and the server. With this technique they obtained a response time improvement for large-sized files by 2.5% to 10% additional to the SRPT technique that takes into account only the size of the file. In the SWIFT algorithm implementation, they assumed that the HTTP requests are embedded with the RTT in their trace driven experiment. This assumption is not a realistic scenario for measuring the RTT "on-the-fly".

2.1.2 Web sever scheduling simulation studies:

In addition to implementation studies, there are simulation studies of scheduling algorithms for web servers [16], [23] M. Gong and C. Williamson [23] identify two different types of unfairness: endogenous unfairness which is caused by an inherent property of a job, such as its size. This aspect of unfairness is invariant. And exogenous unfairness caused by external conditions, such as the number of other jobs in the system, their sizes, and their arrival times. They then continue to evaluate SRPT and other policies with respect to these types of unfairness.

C. Murta and T. Corlassoli [16] introduce and simulate an extension to SRPT scheduling called FCF (Fastest Connection First) that takes into consideration the wide area network (WAN) conditions, such as variability in the bandwidth, round trip time, and packet loss characteristics, in addition to request size when making scheduling decisions. This scheduling policy gives higher priority to HTTP requests for smaller files issued through faster connection. This work is done only by simulation without providing a clear idea on how to

implement it in a real web servers.

E. Friedman et al. [24] propose a new protocol called FSP (Fair Sojourn Protocol) for use in web servers. FSP orders the jobs according to the processor sharing (PS) policy and then gives full resources to the job with the earliest PS completion time. The FSP is a modified version of SRPT and it has been proven through analysis and simulation that FSP is always more efficient and fair than PS given any arrival sequence and distribution. Their simulation results show that FSP performs better than SRPT for large requests, while the SRPT is better than FSP for small requests.

Finally, D. Lu et al. [25] studied the central assumption that is usually made in implementing size-based policies (SRPT, FSP) in a web server . The assumption is the response time of a request is strongly correlated with the size of the file it serves. They found the correlation is weak on a typical web server and the file size may not be a good estimator of overall response time.

2.1.3 Web sever scheduling theoretical studies:

The queuing theory is an old area of mathematics that provides the tools needed for design/analysis of scheduling algorithms in general. Here we will talk only about the web server scheduling theoretical studies that are based on the optimality of SRPT in terms of mean response time (sojourn time) [17], [18], [19].

Recent developments in web server scheduling had led to renewed interest in SRPT queuing. Harchol-Balter et al. [8], [21], [22] proposed the usage of SRPT in web servers to improve the client's response time.

N. Bansal and M. Harchol-Balter [13] compare the SRPT policy and the PS policy analytically for an M/G/1 queue[1] with job size distributions which are modeled by a Bounded Pareto distribution. They show that with link load $\rho = 0.9$[2], the large jobs perform better

[1] From Queueing Theory, theory deals with problems which involve queuing (or waiting):
$M/G/1$ queue has
M(memoryless) : Poisson arrival process
G (General) : general holding time distribution
1 : single server

[2] $\rho = \frac{Average\ Utilization\ of\ the\ Link}{Maximum\ Link\ Capacity}$

under the M/G/1 SRPT queue than the M/G/1 PS queue. And then they prove that for load $\rho = 0.5$, the SRPT improves performance over PS with respect to mean response time for every job, including the very largest job, and for every job size distribution. Also, Harchol-Balter et al. show some theoretical results. For the largest jobs, the slowdown (response time divided by job size) under SRPT is only slightly worse than under PS [28].

In recent work by Bansal [26] and Bansal and Gamarnik [27], interesting results on the mean response (sojourn) time in heavy traffic were obtained that show that SRPT performs significantly better than FIFO if the system is under heavy traffic.

2.2 Admission control and service differentiation

Scheduling techniques can be used to improve response time to higher priority requests. However, in a server overload case, other techniques may apply. Admission control is used to reduce the server workload, by limiting the number of accepted requests, so that the server does not overload. On the other hand, service differentiation is based on classifying different types of clients and giving them different levels of quality of service (QoS), and thus, the resources are allocated to higher priority requests. Usually, both these techniques are combined together.

In [29], the proposed technique concentrates on admission control for Secure Socket Layer (SSL) sessions. This technique depends on the observation that new SSL sessions require a lot of computation, due to the negotiation of full SSL handshakes. And thus, the resumed SSL sessions take higher priority than new SSL sessions. Chen et al. [30] proposed an admission control technique to limit the number of accepted requests based on estimated response time. This approach is only done by simulation.

Other work deals with managing service performance under overloaded dynamic content servers which is based on staged event-driven architecture (SEDA) [31] to break up a complex service in multiple stages connected by queues. Admission control can be performed in these queues based on monitoring the performance of the service (response time) in the stage. In [32], the authors proposed a new method for admission control and request scheduling for e-commerce web sites. The method considers the execution costs of

requests online, distinguishing between different request types to make overload protection mechanism.

In [33] the clients were categorized based on connectivity quality between client and server. Better-connected clients receive higher quality response. On the other hand, users with poor connectivity receive less quality. For example, if connectivity quality is bad for one client, the server selects a lower quality image to send to the client, and it can even select not to send any image, but only text to improve the response time.

Our solution is based on a SRPT scheduling that takes into consideration the WAN parameters in addition to the size of the requested file and also takes into account the interaction of the TCP communication in the implementation to improve the mean response time as a main metric without sacrificing in the throughput.

CHAPTER III

IMPLEMENTATION OF SRRT

In this chapter we describe what we need to build SRRT and SRPT web servers based on Apache running on Linux. Basically two things are needed. First, we need to set up several priority queues at the Ethernet interface. Second, we need to modify the Apache source code to assign priorities to the corresponding requests. In section 3.1 we describe how traffic control queuing discipline achieves priority queuing in the Linux operating system. In section 3.2 we describe how Apache web server code assigns and updates priorities to the requested static files. In section 3.3 we give an overview of TCP congestion control and how SRRT implementation is informed by certain parameters of the TCP protocol.

3.1 Differentiated services and traffic control in Linux

Linux has Quality of Service (QoS) mechanism called "Traffic Control"[1]. Traffic control provides a set of queuing systems and priority schemes between the IP layer and the network device to condition network traffic. The traffic control tool (tc) enables the user to control these queues and the queuing mechanisms of packets transmitted and received over the networked device. Therefore, the typical uses of traffic control are to raise the priority of some kind of traffic higher than others, or to limit the rate at which traffic is sent, or to block undesirable traffic.

There are three main elements of traffic control: classifier, scheduler, and queues. Packets are classified by looking at the packets content or at other information related to the packets and then classifying them into different classes. Packets are then placed into distinct queues, and eventually scheduled for transmission. The class of a packet determines in

[1] The support for differentiated service (DiffServ) is integrated into 2.4 Linux kernels and above by default. If the DiffServ is not supported, it can be added by enabling "QoS and/or fair queuing" kernel configuration options in the section "Networking Options" where configering the kernel source. More clear explanation of how to configure Linux to implement the DiffServ is given in [37].

which queue the packet goes and how it is scheduled [38]. Figure 2 illustrates this process.

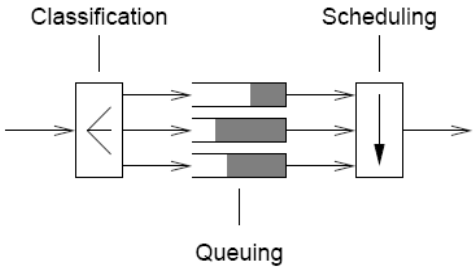


Figure 2: Traffic contol elements [38]

The queuing disciplines (*qdisc*) form a basic block for supporting QoS in Linux. There are many queuing disciplines that are supported in Linux which includes: Packet First-In, First-Out (*pfifo_fast*), Priority (*prio*), Token Bucket Flow (*tbf*), Stochastic Fair Queuing (*sfq*), Class Based Queue (*cbq*), and others. All types of queuing disciplines are explained in detail in [40]. Here we are interested in the *pfifo_fast qdisc* which is the default queuing discipline in Linux. Also, we are interested in *prio qdisc* which is needed to implement our algorithm.

3.1.1 Default Linux qdisc

The default qdisc under Linux is the *pfifo_fast qdisc*. This *qdisc* is slightly more complex that FIFO (First-In, First-Out) as, it provides some prioritization. This *qdisc* has three different queues (bands). Within each band, FIFO rules apply. The highest priority traffic is placed into band 0; as long as there are packets waiting in band 0, band 1 will not be processed. The same goes for band 1 and band 2. The *pfifo_fast qdisc* is shown in Figure 3.

The data being passed from user space is stored in socket buffers corresponding to each connection. When data streaming passes from the socket buffers to TCP layer and IP layer, the TCP headers and the IP headers are added to form packets. The packet flow corresponding to each socket is kept separate from other flows [21]. After that, packets are sent from IP layer to *pfifo_fast* queuing discipline.

pfifo_fast qdisc is a classless queuing discipline, so it cannot be configured. The packet priorities are determined by the kernel according to the so called Type of Service (TOS) flag

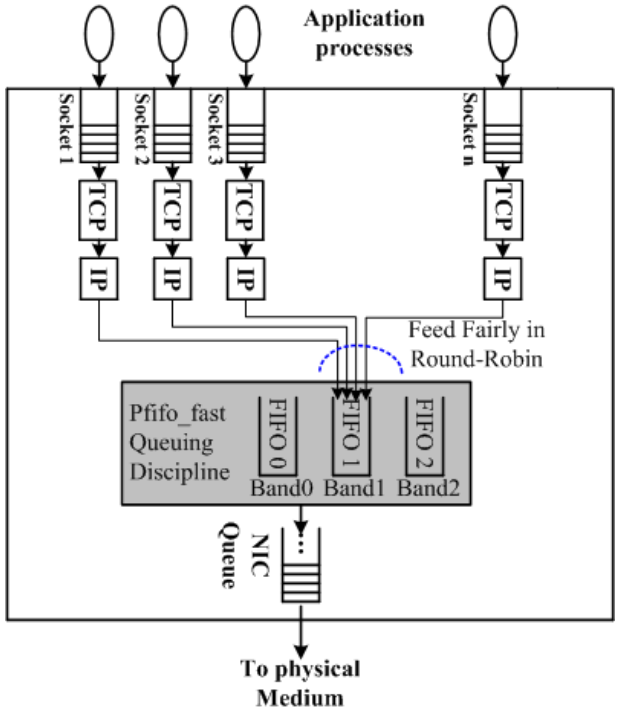


Figure 3: Defualt Linux qdisc(pfifo_fast)

and priority map (priomap) of packets[2]. However, all packets using the default TOS value (0x00) are queued to the same band. So the three bands appear as a single FIFO queue in which all streams feed in a Round-robin service (Processor Sharing (PS)): *Fairness per flow, all requests from processes or threads are given an equal share of capacity.* Packets leaving this queue drain in a network device (NIC) queue and then out to the physical medium (network link).

Now, to see which queuing discipline is in effect, the traffic control[3] (*tc*) command in the user space can be used as follows:

```
# tc qdisc
```

The above command will generate the following:

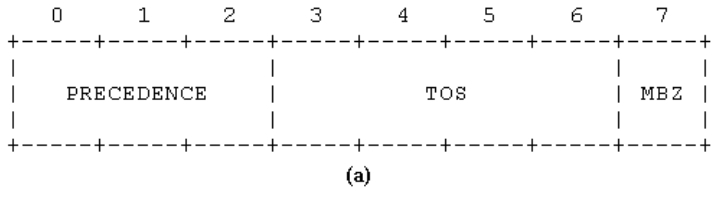
[2]For an exact description of how packet priorities are determined, see [40] section 9.2.1.

[3]To check the current working version of iproute2 package is available in Linux:, issue the following command:

```
# tc -V
tcutility, iproute 2 -ss040831
```

qdisc pfifo_fast 0: dev eth0 bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1

As shown above, the default qdisc on eth0 device is *pfifo_fast* with 3 priority bands and a FIFO ordering in each band. The *priomap* determines how packet priorities, as assigned by the kernel, map to bands. Mapping occurs based on the TOS value of the IP header. The TOS field is shown in Figure 4(a) and the meaning of TOS bits in Figure 4(b). The TOS bits can be combined to form 16 values (2^4) for the TOS field which are given in the Figure 4(c). Using the default *priomap*, packets with a TOS value of 0x10, 0x12, 0x14 or 0x16 will be sent to band 0. Packets with a TOS value of 0x0, 0x4, 0x6, 0x18, 0x1a, 0x1c, 0x1e to band 1 and packets with a TOS value of 0x2, 0x8, 0xa, 0xc and 0xe to band 2. See [40] for further details.



(a)

Binary	Decimcal	Meaning
1000	8	Minimize delay (md)
0100	4	Maximize throughput (mt)
0010	2	Maximize reliability (mr)
0001	1	Minimize monetary cost (mmc)
0000	0	Normal Service

(b)

TOS	Bits	Means	Linux Priority	Band
0x0	0	Normal Service	0 Best Effort	1
0x2	1	Minimize Monetary Cost	1 Filler	2
0x4	2	Maximize Reliability	0 Best Effort	1
0x6	3	mmc+mr	0 Best Effort	1
0x8	4	Maximize Throughput	2 Bulk	2
0xa	5	mmc+mt	2 Bulk	2
0xc	6	mr+mt	2 Bulk	2
0xe	7	mmc+mr+mt	2 Bulk	2
0x10	8	Minimize Delay	6 Interactive	0
0x12	9	mmc+md	6 Interactive	0
0x14	10	mr+md	6 Interactive	0
0x16	11	mmc+mr+md	6 Interactive	0
0x18	12	mt+md	4 Int. Bulk	1
0x1a	13	mmc+mt+md	4 Int. Bulk	1
0x1c	14	mr+mt+md	4 Int. Bulk	1
0x1e	15	mmc+mr+mt+md	4 Int. Bulk	1

(c)

Figure 4: priomap as assigned by the kernel [40]

3.1.2 prio qdisc to implement SRRT

To implement our scheduling algorithm, we need several priority queues. And this can be achieved by *prio qdisc*. The *prio qdisc* has similar behavior like *pfifo_fast* but can be configured. The *prio qdisc* works on a very simple principle. When it is ready to dequeue a packet, the first band (queue) is checked for a packet. If there is a packet, it gets dequeued. If there is no packet, then the next band is checked, until the queuing mechanism has no more classes to check. Figure 5 shows the *prio* queuing discipline to implement SRRT and SRPT.

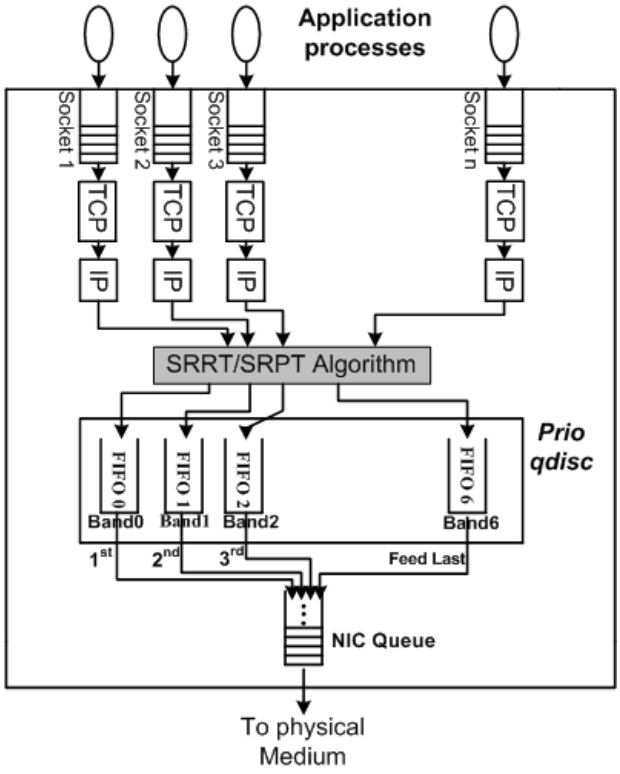


Figure 5: *prio qdisc* to implement SRRT and SRPT

Three methods are available for *prio qdisc* to determine in which band a packet will be enqueued:

- *From user space*: A process with sufficient privileges can encode the destination class directly with socket option (*SO_PRIORITY)*. We will use this method in the SRRT implementation by giving the Apache code the responsibility to assign the priorities

to the corresponding connection.

- *With the priomap:* Based on the packet priority, which in turn is derived from the Type of Service assigned to the packet.
- *With a tc filter:* A *tc* filter attached to the root *qdisc* can point traffic directly to a class.

3.1.2.1 ***Setup the traffic control tree***

As seen above, it is needed to create a traffic control tree as shown in Figure 6 to enable priority queue with seven leaves.

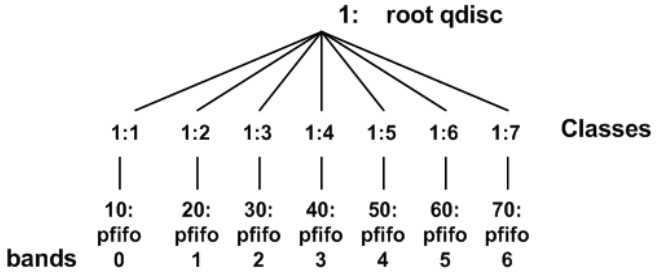


Figure 6: Traffic control tree

The *tc* control command can be used to setup the above traffic control tree to create several priority queuing discipline as shown in the following script.

```
#!/bin/bash
tc qdisc add dev eth0 root handle 1: prio bands 7 priomap 0 1 2 3 4 5 6 7 8 9 10
11 12 13 14 15
tc qdisc add dev eth0 parent 1:1 handle 10: pfifo
tc qdisc add dev eth0 parent 1:2 handle 20: pfifo
tc qdisc add dev eth0 parent 1:3 handle 30: pfifo
tc qdisc add dev eth0 parent 1:4 handle 40: pfifo
tc qdisc add dev eth0 parent 1:5 handle 50: pfifo
tc qdisc add dev eth0 parent 1:6 handle 60: pfifo
tc qdisc add dev eth0 parent 1:7 handle 70: pfifo
```

Here we have 16 priority bands (queues) and we want to use only the bands that range in number from 0 to 6, where band 0 has highest priority and band 6 has lowest priority.

The *prio qdisc* works as follow: the priority queues feed in a prioritized fashion into the network device queue. When the *prio qdisc* is ready to dequeue a packet, band 0 is checked for a packet. If there is a packet, it gets dequeued. If there is no packet, then band 1 is checked, and so on until reaching band 6 (last band in our example).

Configuration check

Now, the configuration has been completed. To print the current parameter settings statistics, issue the *show* option of the *tc* command. The *-s* option means statistics, and *-d* option means detail.

Let us examine what we have created.

```
# tc -s -d qdisc show dev eth0
qdisc prio 1: bands 16 priomap 0 1 2 3 4 5 6 0 1 1 1 1 1 1 1 1
Sent 0 bytes 0 pkts (dropped 0, overlimits 0 requeues 0)
Sent 0 bytes 0 pkts (dropped 0, overlimits 0 requeues 0)
qdisc pfifo 10: parent 1:1 limit 1000p
Sent 0 bytes 0 pkts (dropped 0, overlimits 0 requeues 0)
qdisc pfifo 20: parent 1:2 limit 1000p
Sent 0 bytes 0 pkts (dropped 0, overlimits 0 requeues 0)
qdisc pfifo 30: parent 1:3 limit 1000p
Sent 0 bytes 0 pkts (dropped 0, overlimits 0 requeues 0)
qdisc pfifo 40: parent 1:4 limit 1000p
Sent 0 bytes 0 pkts (dropped 0, overlimits 0 requeues 0)
qdisc pfifo 50: parent 1:5 limit 1000p
Sent 0 bytes 0 pkts (dropped 0, overlimits 0 requeues 0)
qdisc pfifo 60: parent 1:6 limit 1000p
Sent 0 bytes 0 pkts (dropped 0, overlimits 0 requeues 0)
qdisc pfifo 70: parent 1:7 limit 1000p
Sent 0 bytes 0 pkts (dropped 0, overlimits 0 requeues 0)
```

To return to the default Linux qdisc, remove the root qdisc by issuing the following command:

```
#tc qdisc del dev eth0 root
```

3.2 Modifications to Apache web server to implement SRRT

After enabling the priority queuing in Linux by using *tc* command, we need to assign priorities to the packets and put them in the designated queue. The task of prioritizing sockets is given to the Apache web server code. We made changes to the Apache HTTP Server 2.2.4 code to prioritize connections. The modifications are fairly isolated to two specific files: httpd-2.2.4/server/protocol.c and httpd-2.2.4/server/core.c. The *diffs* for these files can be seen in Appendix B. Apache uses the *setsockopt(...,SOL_SOCKET, SO_PRIORITY, ...)*[4] function to set the priority to the sockets. Apache sends the requested file, and after every IOBUFSIZE[5](8KB), the remaining size of the file decreases. A check is made whether the priority level of the remaining size has fallen below the threshold for the current priority class. If it is so, Apache updates the socket priority with another call to *setsockopt()*.

The installation of the SRRT- and the SRPT-modified Apache servers are exactly the same as the installation of standard Apache. The only thing that might need to change when experimenting with SRRT and SRPT servers are the priority array values to determine the priority class of the socket. The priority array defined in httpd-2.2.4/server/protocol.c in the following lines.

For SRPT:

```
unsigned int prios[7] = { 0, 1000, 2000, 5000, 15000, 80000, 1000000000 };
```

These values represent the file sizes classes. Files with size less than 1000Byte go to band 0 and files with size between 1000Byte-2000Byte go to band 1 and so on. These values (*cutoffs*) depend on the file size distribution of the files served by the Apache web server, which are chosen based on the rule of thumb given in [21] to prevent the starvation in **heavy**

[4]For more details about sockets options (getsockopt and setsockopt) see [61].
[5]IOBUFSIZE: the size of the server's internal read-write buffers

tailed distributions[6] of file sizes. By denoting the cutoffs by $x_1 < x_2 < ... < x_n$, then the lowest cuttoff x_1 should be such that about 50% of the requests have size smaller than x_1. The highest cuttoff x_n should be low enough such that $0.5\% - 1\%$ of the requests have size $> x_n$ to prevent starvation of large requests. A logarithmic spacing can be taken for middle cutoffs.

For SRRT:

```
unsigned int prios[7] = { 0, 100, 500, 1000, 20000, 80000, 1000000 };
```

These values are taken depending on the response time in milliseconds. The upper value is set to 1000000 msec to prevent starvation of long response. The 80000 corresponds to estimated response time for large file, 2MB, through bad network conditions ($RTT = 350ms$, $cwnd = 3$). Assuming that the Maximum Segment Size $MSS = 1460$Bytes, the approximated response time is

$$Response\_time = RTT\left(\frac{File}{cwnd \times MSS} + 1\right)^{7}$$
$$Response\_time = 350(\frac{1 \times 1024 \times 1024}{3 \times 1460} + 1) = 84140.0ms$$

The other values are taken based on the experiment trials. After experimenting with different values, we found these cutoffs gave us good results.

There are two problems caused by giving the web server code the responsibility for prioritizing the connections. The first one is the overhead of the system calls to assign priorities, and the second is the need to modify the web server code which is limited to a few lines in a handful files as seen in Appendix B. The first problem is alleviated by limiting the number of *setsockopt()* calls that have to be made. Typically only one call is made per connection. In the worst case, the number of *setsockopt()* calls equal the number of

[6] The heavy-tailed distributions are probability distributions with infinite variance. The distribution of a random variable X is said to have a heavy tail if: $\Pr\{X > x\} \tilde{} x^{-\alpha}, \quad 0 < \alpha < 2.$

[7] See section 3.3 for more detials

priority classes (6 in our experiments) per connection. The implementation of SRRT using *prio qdisc* limits us to 16 priority bands. However, we used only 6 priority bands in our experiment.

TCP SYN-ACKs gets by default into the highest priority band "band 0". Here, we will take into consideration the recommendation given by [21]. Because the start up of the connection is an essential part of the total response delay, especially for small files before the size of the file is known, no sockets are assigned to priority band 0, but are assigned to other bands of lower priority, to prevent packets sent during the connection start up waiting in a long queue. SYN packets are 40 bytes (320bits), so the SYN-ACKs constitute a negligible fraction of the total load. Thus assigning them to higher priority does not affect the performance.

3.3 Using TCP protocol information to implement SRRT

To address all the client-server interaction on the Internet such as variability in bandwidth, round trip time and packet loss in addition to file size, we propose a simple and low-overhead estimator for response time that make scheduling perform better. We propose the SRRT algorithm that takes into consideration the WAN parameters: delay, loss rate and TCP throughput by benefiting from TCP implementation.

3.3.1 Data transmission over TCP

To communicate over a TCP protocol [49], [50], [51], [52], [53], [54] the receiver informs the sender about the available buffer space at the receiver using the TCP header field "*window*". The window size is the amount of data a sender can send before it gets an acknowledgment (ACK) back from the receiver. No more than this amount of data should be transmitted in the network by this TCP connection. Also the sender must buffer the sent data until it

has been acknowledged by the receiver, so that the data can be retransmitted if needed to recover from an error. For each ACK at the sender, sent data is dropped from the window and a new segment fills the window.

The main reason for using the sender window is congestion control. Due to TCP's congestion control mechanism, TCP widow sizes can be bound to the maximum theoretical throughput rate ($\frac{cwnd \times MSS}{RTT} bps$) despite the actual bandwidth capacity of the network path. Too small a TCP window size can degrade the network performance ($rate << capacity$), and too large window size causes the rate to be greater than the bandwidth capacity which is leads to congestion in the network.

3.3.2 TCP congestion control

TCP congestion control and window size adjustment are essentially based on the TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery algorithms [53].

3.3.2.1 Slow start

TCP operates over a heterogeneous Internet. TCP has no advance knowledge of network conditions, thus it has to adapt its behavior according to network current state. To avoid that a starting TCP connection inject data at a rate higher than the network can handle, a Slow Start mechanism was introduced into TCP. The Slow Start effectively probes to perceive the available network bandwidth. Slow Start starts slowly, and then increasing its window size as TCP gains more confidence in the network's ability to handle traffic.

Slow Start implementation uses two values: congestion window (*cwnd*) and Slow Start Threshold (*ssthresh*). The *cwnd* is a transmit window limit at the sender end. *ssthresh* is the threshold for determining the point at which TCP exits slow start. TCP connections start with *ssthresh* set to 64KB. Initially, the *cwnd* value is set to maximum segment size

(MSS)[8]. Every time an ACK arrives, the *cwnd* window is incremented by one MSS. The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the *cwnd* is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, the *cwnd* is increased to four. Thus, the *cwnd* is effectively doubled per RTT. If *cwnd* increases beyond *ssthresh*, the TCP congestion control mode is changed from Slow Start to Congestion Avoidance. Exiting slow start signifies that the TCP connection has reached an equilibrium state where the *cwnd* closely matches the networks capacity.

3.3.2.2 Congestion avoidance

Once *cwnd* is greater that *ssthresh*, TCP enters the congestion avoidance mode. From this point on, TCP *cwnd* will be much more conservative. The *cwnd* will grow linearly and not exponentially. In this mode, the primary objective is to maintain high throughput without causing congestion. The *cwnd* continues to use a linear growth until congestion is detected. If TCP detects packet loss, it assumes that congestion has been detected over the Internet. As a corrective action, TCP reduces its data flow rate by reducing *cwnd* to go back to slow start.

3.3.2.3 Fast retransmit and fast recovery

TCP Slow Start and Congestion Avoidance lower the data throughput drastically when segment loss is detected. Fast Retransmit and Fast Recovery have been designed to speed up the recovery of the connection, without compromising its congestion avoidance characteristics. Fast Retransmit: TCP receives duplicate acks and it decides to retransmit the segment, without waiting for the segment timer to expire. This speeds up recovery of the

[8]MSS refers to the size of the biggest chunk of data that can be sent in a single TCP segment. Typical MSS values: 1460 bytes, 536 bytes (default), and 512 bytes.

lost segment. Fast Recovery: Once the lost segment has been transmitted, TCP tries to maintain the current data flow by not going back to slow start. TCP also adjusts the window for all segments that have been buffered by the receiver.

3.3.3 TCP time-out and round trip time

The TCP congestion control mechanism involves Time-outs that cause retransmissions. A timer is started after a segment is sent, and retransmit will occur if the timer times out before data in the segment has been acknowledged. Therefore, it is important that hosts have an accurate Time-out method. The Time-outs are set as a function of RTT. To compute the RTT, TCP sets the timestamp t_s with the sent packet and when its ACK arrives, record timestamp t_a.

$$SampleRTT = t_a - t_s$$

To compute Time-out, there are two algorithms:

Algorithm 1 *Karn/Partridge Algorithm: An adaptive retransmission algorithm [59]:*

$EstimatedRTT = \alpha \times Old\_EstimatedRTT + (1 - \alpha) \times New\_SampleRTT$

$TimeOut = \beta \times EstimatedRTT$

In the event of a retransmission due to a time-out,

$New\_TimeOut = \gamma * TimeOut$

where α, β, and γ values usually 0.9, 2, and 2 respectively.

Algorithm 2 *Jaconson/Karels Algorithm [60]:*

This algorithm takes into account the mean and the variance of RTT.

$Difference = New\_SampleRTT - Old\_EstimatedRTT$

$EstimatedRTT = Old\_EstimatedRTT + \delta \times Difference$

$Deviation = Old\_Deviation + \rho(|Difference| - Old\_Deviation)$

In the event of a retransmission due to a time-out,

$TimeOut = EstimatedRTT + \eta\ Deviation$

where δ, ρ and η are recommended to be $\frac{1}{2^3}, \frac{1}{2^2}$ and 4 respectively

$New\_TimeOut = \gamma * TimeOut$ is still applied.

3.3.4 Approximating the remaining response time for the TCP connection

After processing a HTTP request, the server code uses the TCP_INFO socket option to get useful information that will be used to estimate the remaining response time of the request on the server side. By using *getsockopt(...,IPPROTO_ TCP, TCP_ INFO,...)* the server can know the RTT and TCP throughput for each connection. The requested file size is already known by the server after processing the request. Hence the remaining response time (RRT) can be approximated as follows:

$$RRT(\text{sec}) \approx RTT + \frac{RFS(byte)}{R(bytes/sec)}$$

where RFS is remaining length of the requested file and R is the approximated TCP transfer rate.

The TCP window size is the most important parameter for achieving maximum transfer rate (throughput) across the network. Due to TCP's congestion control mechanism, the TCP protocol limits the use of physical resources. TCP *cwnd* size can limit the maximum transfer rate regardless of the actual physical bandwidth of the network path. To achieve the maximum throughput, the TCP window should be greater than the bandwidth-delay product.

$$WindowSize \geqslant Bandwidth(bytes/sec) \times RoundTripTime(sec)$$

$$cwnd \geqslant BW((bytes/sec) \times RTT(\text{sec})$$

$$BW(bytes/sec) \leqslant \frac{cwnd(byte)}{RTT(\text{sec})}$$

Roughly, the current maximum transfer rate (R) for a connection can be approximated

by:

$$R(bytes/sec) = \frac{cwnd(byte)}{RTT(\text{sec})}$$

3.3.5 The final SRRT algorithm

After processing a request we know its file size, the RTT and an estimate of the TCP throughput. Collecting this information is simple and requires little work per request. The SRRT can be summarized in the following steps:

1. During the connection setup phase, a request respected to socket with the highest priority (priority 0)

2. Get the following quantities for the TCP by using *getsockopt(..., IPPROTO_ TCP, TCP_ INFO,...)* function:

   (a) Round trip time (RTT)

   (b) Congestion Window ($cwnd$)

3. Use the congestion window and the RTT values to estimate the service rate (R) for each connection:

   $$R(bytes/sec) = \frac{cwnd(byte)}{RTT(\text{sec})}$$

4. Determine the Remaining File size (RFS)

5. For each connection, estimate the request's Remaining Response Time (RRT)

$$\begin{aligned} RRT(\text{sec}) &= RTT + \frac{RFS(byte)}{R(bytes/sec)} \\ &= RTT(\text{sec}) + \frac{RFS(byte)}{cwnd(byte)} \times RTT(\text{sec}) \end{aligned}$$

$$\boxed{RRT(\text{sec}) = RTT(\text{sec}) \left(1 + \frac{RFS(byte)}{cwnd(byte)}\right)}$$

6. Based on RRT, the priority of the socket corresponding to the each request is determined. The priority of the sockets are updated dynamically after every IOBUFSIZE (8192bytes), the size of the server's internal read-write buffers, of the sent file, by repeating steps 2 to 5 above.

As seen above, our algorithm for the estimating the remaining response time depends on three variables; remaining length of the sent file, current Round Trip Time, and the current TCP congestion window size. And thus, we consider almost all aspects that affect data transfer over the Internet since the RTT and the congestion window size change dynamically according to network conditions. The estimated RRT is influenced by network conditions, and we obtain updated estimates for this RRT with each iteration of steps 2 to 5 in the algorithm above. The highest priority is given to the connection that has the best performance estimate: the connection that needs to transfer small file through an uncongested path, which has smaller RTT and large Congestion Window. As a consequence, by giving priority to the fast connection which will finish faster, we release resources at the server to enable other requests waiting in the queue to use them.

In addition to the above algorithm, we tried out with another algorithm which is based on the total estimated response time (TERT) and not only on the estimated remaining response time (RRT). In this algorithm, the priorities are given to the requests based on the shortest total estimated response time (STERT). STERT gives better results than SRPT but worse than SRRT. Accordingly, in all coming discussion we will talk only about SRRT. Appendix A contains the details for STERT with some results.

CHAPTER IV

EXPERIMENT SETUP

This chapter discusses how to set up our experiment and how to go about benchmarking web servers running on Linux.

4.1 Benchmarking web server

The aim of our experiment is to minimize the mean response time from the server to clients without affecting other performance metrics such as the number of requests per second the server is able to sustain, the number of bytes transferred per second (data throughput) and the number of simultaneous connections the web server can handle without error.

A typical network layout to conduct web server benchmarking requires the following components:

- A server running the web server software under test
- A number of clients running load generating software
- A network connecting the clients to the server

Figure 7 shows a typical network layout for web server benchmarking. A number of clients are connected via a switch to the server under test.

4.2 The bottleneck

For benchmark results to be useful, it is critical to determine where of the performance bottleneck in the system is. The bottleneck could be on:

- The client machines

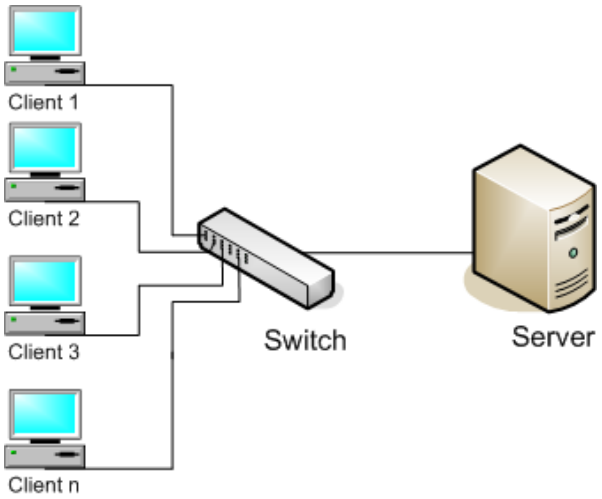


Figure 7: Network layout for Web server benchmarking

- The network bandwidth
- The server

4.2.1 Network connection

On a web server servicing primarily static files, network bandwidth is the most likely source of a performance bottleneck [21], [43]. Even modest web servers supply enough data to completely saturate a T3 connection (45Mbps) or a 100Mbps Fast Ethernet connection [46]. As the network is the bottleneck resource, our scheduling policy for static contents is applied on the access link out of the web server. Thus, the system load can be defined as the link utilization.

However, we represent the system load by the number of concurrent users (User Equivalents, UEs) generated by the workload generator. As the number of UEs increases the average use of the link bandwidth increase. We found, about 1000 UEs can saturate the 100Mbps link. So we can consider the $load = 1$ at 1000 UEs for 100Mbps link.

4.2.2 Client machines

It is critical to prevent client machines from becoming the bottleneck. To eliminate this possibility, several clients are used. To determine whether or not the clients are the bottleneck, the same total number of requests are run twice over different numbers of clients. If the results change, then the client(s) is the bottleneck, and the number of clients should be increased. We also use six client machines to represent different networks with different RTTs and loss rates to emulate clients from heterogeneous WANs. In our experiments, a client machine CPU utilization did not exceed 7% and memory usage was less than 30%, so clients were not the bottleneck.

4.2.3 The server

A single server can support several thousand concurrent users depending on system specifications. In May 2005, a national mirror server for Ireland, *ftp.heanet.ie* (single Apache web server), was sustaining more that 20000 simultaneous HTTP downloads and had saturated a gigabit of connectivity [47]. ftp.heanet.ie was running with Debian GNU/Linux 3.0.The hardware was a 667 MHz CPU, 1.5GB of memory, 30 GB of storage. Therefore, the assumption here the server is not the bottleneck.

4.3 Experiment setup

Our experimental setup consists of seven machines connected by 10Mbps switch in the first experiment and by 100Mbps Fast Ethernet connection in the second experiment. Each machine has an Intel Pentium 4 CPU 3.20 GHz, 504 MB of RAM. We used the Linux 2.6.18 operating system. One of the machines (the server) runs Apache 2.2.4. The other machines act as web clients. The client machines generate loads using the Scalable URL Request Generator (SURGE) [34]. On each client machine, Network Emulator (*netem*) is used to

emulate the properties of a Wide Area Network (WAN).

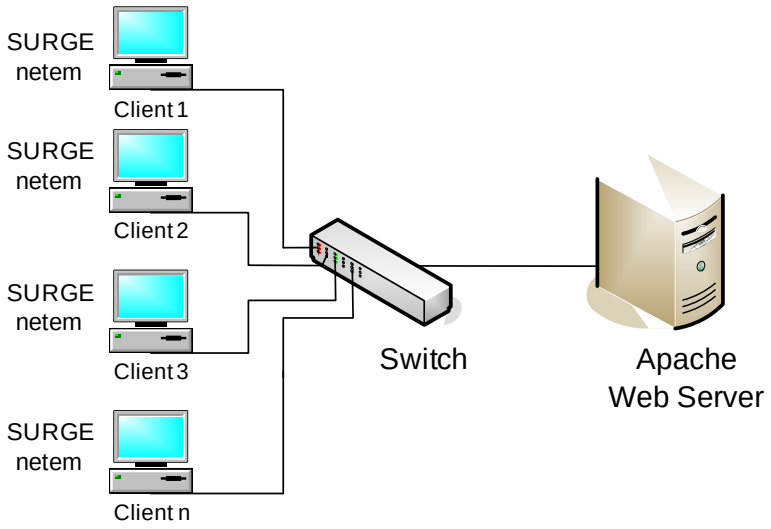


Figure 8: Experiment setup

4.4 Workload traffic generators

Workload generators are programs that run on client machines to emulate the web client behavior by constructing HTTP requests and sending them to the server. The main task of a workload traffic generator is to help understand the performance of web servers. A workload generator can typically change the number and the distribution of requests that it generates (load) and measure how the server responds to the load variations.

4.4.1 Performance metrics

There are many performance metrics to measure such as:

1. *Mean response time:* let a testing client send requests at a certain rate and then measure the average time needed to receive a response.

2. *Throughput:* the amount of data a server can generate per second.

3. *Latency for Reading a disk file:* the time needed for reading a file from the server.

4. *Other qualities:* fairness and robustness.

The metric that has received the most interest in web server scheduling policies is mean response time, [10], [16], [21], [22], [25], because user perceived performance is a function of the response time. Therefore, mean response time is considered as the main performance metric in our experiments.

4.4.2 Workload generators

Many workload traffic generators exist for evaluating web server performance. SPECweb99 [35] is a tool distributed by the System Performance Evaluation Cooperative (SPEC) organization. This benchmark is probably the most commonly used and most cited tool. Unfortunately, this tool costs money. Httperf [36] is another frequently used workload generator. Httperf is an open source benchmark utility from HP labs. This tool provides a flexible facility for generating various HTTP workloads and measuring server performance. However, it does not provide request size distribution.

Many other tools exist, including WebStone, TPC-W, WaspClient, S-Client, WAGON, WebBench, and SURGE [34] (Scalable URL Reference GEnerator). SURGE is a well-known workload generator which generates references based on empirically derived distributions, server file size distribution, request size distribution, relative file popularity, embedded file references, temporal locality of reference, and idle periods ("think times") of individual users. Thus, we use the SURGE to understand how servers respond to variation in load.

4.4.3 SURGE: Workload generator for web traffic

SURGE generates representative workload for static web services. It exercises servers and networks by generating a stream of HTTP requests that mimics the behavior of users executing web applications. SURGE generates a high variability in the workload. The

SURGE tool is composed of two basic parts: one concerning the concept of user equivalents and the other concerning a set of distributional models.

The concept of user equivalents (UEs) states that the workload generated by SURGE should imitate the structure of the real traffic generated by different users. In particular, SURGE reproduces the web requests of these users (see Figure 9).

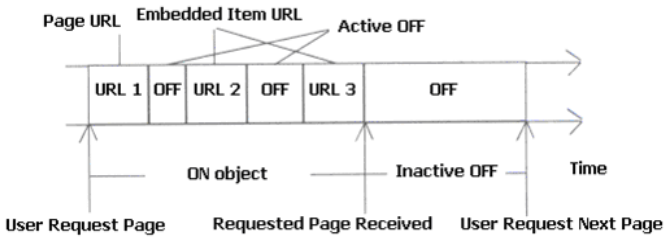


Figure 9: Representation of a request form [32]

Each user equivalent is defined as a single process in an endless loop that alternates a time period for making requests for web files, and lying idle. Both the web request and idle times must show the distributional and correlation properties characteristic of real web users. Thus each UE is an ON/OFF process where ON refers to periods during which data is being requested and OFF to those during which nothing happens (idle times).

The distributional models express the set of probability distribution used by each UE. SURGE generates references matching the following set of statistical properties for web streams: Server File Sizes distribution, Request Sizes distribution, Relative files Popularity, Embedded file References, Temporal Locality of reference and OFF Times or Idle periods of individual users (See the Table 1).

1. The "File Sizes" distributional model describes the set of files stored on the server that must agree in distribution with empirical measurements. We have 2000 different

Component	Model	Probability Density Function	Parameters
File Sizes- Body	Lognormal	$p(x) = \frac{1}{2\sigma\sqrt{2\pi}} e^{-(\ln x - \mu)^2/2\sigma^2}$	$\mu = 9.357; \sigma = 1.318$
File Sizes- Tail	Pareto	$p(x) = \alpha k^\alpha x^{-(\alpha+1)}$	$k = 1333K; \alpha = 1.1$
Popularity	Zipf		
Temporal Locality	Lognormal	$p(x) = \frac{1}{2\sigma\sqrt{2\pi}} e^{-(\ln x - \mu)^2/2\sigma^2}$	$\mu = 1.5; \sigma = 0.80$
Request sizes	Pareto	$p(x) = \alpha k^\alpha x^{-(\alpha+1)}$	$k = 1000; \alpha = 1.0$
Active OFF Times	Weibull	$p(x) = \frac{bx^{b-1}}{ab} e^{-(x/a)^b}$	$a = 1.46; b = 0.382$
Inactive OFF Times	Pareto	$p(x) = \alpha k^\alpha x^{-(\alpha+1)}$	$k = 1; \alpha = 1.5$
Embedded References	Pareto	$p(x) = \alpha k^\alpha x^{-(\alpha+1)}$	$k = 1; \alpha = 2.43$

Table 1: Distributions used in the workload generator from [32]

file sizes on the server.

2. The "Request Sizes" distributional model describes the collection of files sizes transferred over the network according to empirical size measurements. In general, this distribution is different from the file size distribution, because some files may be transferred multiple times, while others may never be transferred. This distribution is a Pareto and affects appropriately the network.

3. The "Popularity" distributional model describes the number of requests made to each single file on the server. This probability distribution follows Zipf's Law and is related to the previous two.

   The "Embedded References" distributional model captures the structure of embedded web objects. This probability distribution follows Pareto distribution and describes how many objects (images, texts, sounds, etc.) embedded in each web page are required to display the result correctly. OFF times between embedded references (Active OFF times) are typically shorter than OFF times between web objects themselves (Inactive OFF times). Server logs do not store the number of embedded references fetched for a particular web page.

4. The "Temporal Locality" distributional model describes the probability that, once a file has been requested, it will be requested again in the near future. This probability distribution is modeled using lognormal distribution and exercises the caching effectiveness.

5. The "inactive OFF Times" distributional model is necessary to capture the bursty nature of requests of individual web user's requests, and follows a Pareto distribution.

6. The "active OFF Times" distributional model reproduces the time to transfer web objects. The Weibull distribution gives a good fit with empirical values. Embedded References of a web object are considered to be the sequence of files fetched by a given user for which the OFF time between transfers was less than a given threshold (1sec).

4.5 Network emulation

Web servers have to run over Wide Area Networks (WANs). Thus, any application on web servers must take into consideration real life network delays and packet loss, and other parameters. An application designed only for a LAN environment will not work properly when used over the Internet. Lab environments are not influenced by the real world Internet, thus the motivation for a WAN Emulator is to provide a way to reproduce Internet conditions in a lab environment.

The Internet environment includes:

- Packet loss which results from busy or congested network links with lots of errors.

- High delay comes from carriage of IP datagrams across low speed or long distance links (e.g. in the case of satellite).

- Delay variation comes from the bursty nature of most IP traffic that results in varying

amounts of buffering of data occurring for a given network connection over time.

- The bandwidth limitation in the forward and backward directions comes from the diversity of network technologies such as ADSL which provide more bandwidth in one direction than other.

Many network emulators exist. Nistnet, Dummynet and Netem are network emulation software packages that are similar in design and can be run on the Linux operating system to emulate the properties of wide area networks. Nistnet [42] was developed by the National Institute of Standard and Technology (NIST). Nistnet is a patch for 2.4 Linux kernels that allows packets drops, packets delays and connection bandwidth limiting. Dummynet [44] is a part of the FreeBSD system that simulates network delays and loss by manipulating the IP queue inside the kernel. Netem [45] emulates wide area network delays, packet loss, packet duplication, packet corruption, packet re-ordering and packet rate control.

Netem is built using Quality of Service (QoS) and Differentiated Service (Diffserv) and configured by the command line tool "*tc*" which is already enabled in the Linux kernel 2.6. Netem was used in this work since it is included under the kernel 2.6 and it provides the necessary options to emulate real world networks. Netem is a classful queuing discipline which can add delay, loss, duplication, or reordering of packets as arguments to the tc command.

4.6 Tuning parameters

4.6.1 Tuning Apache web server

Apache's main configuration file is httpd.conf. There are several parameters that can be modified to improve performance. We used the default configuration of Apache, except for the maximum number of connections, by changing MaxClients value. This parameter

sets the upper bound on the number of processes that Apache can have running concurrently. Larger values allow larger numbers of clients to be served simultaneously. Very large values may require Apache to be re-compiled. The default value of MaxClients is 256. We change MaxClients to 2500 to increase the number of simultaneous requests that can be supported by the server. To configure more than 256 clients, we edit the #define HARD_SERVER_LIMIT from 256 to 4096 in httpd.h file and recompile the code.

4.6.2 Tuning the operating system at the server

A web server is supported by the operating system which has a large effect on server performance. Under Linux, it is possible to set certain kernel parameters by using the echo command. By benefit from "TCP Tuning Guide" [48] and in order to achieve better performance we run the following script to set some tuning parameters on Linux.

```
#!/bin/bash
echo "10000000" > /proc/sys/net/core/wmem_max
echo "10000000" > /proc/sys/net/core/rmem_max
echo "10000000" > /proc/sys/net/core/rmem_default
echo "12000" > /proc/sys/net/ipv4/tcp_max_syn_backlog
echo "2000000" > /proc/sys/net/ipv4/tcp_max_tw_buckets
echo "400000" > /proc/sys/net/core/netdev_max_backlog
echo "500000" > /proc/sys/fs/file-max
ifconfig eth0 txqueuelen 30000
```

Increasing the size of the send socket buffer size is done by echo *wmem_max* and increasing the size of the receive socket buffers size is done by echo *rmem_max.* Increasing the amount of memory available to the TCP/IP input queue by setting the default value *rmem_default* and maximum value *rmem_max.* Increasing the number of TCP SYN packets that the server can queue before SYNs are dropped by increasing *tcp_max_syn_backlog* value. And increasing the number of TCP connections that are allowed in the TIME-WAIT

state by *tcp_max_tw_buckets value.* The *netdev_max_backlog value* sets the length for the number of packets that can be queued in the network core (below the IP layer).This allows more memory to be used for incoming packets, which would otherwise be dropped. Increasing the file descriptor limit is done by increasing *file-max* value. Finally, to increase the length of the interface configuration issue *"ifconfig eth0 txqueuelen 30000"* command.

CHAPTER V

RESULTS AND ANALYSIS

In this chapter we summarize the results. The mean response time is taken as the main performance metric from the client's view. We compare the performance of Shortest-Remaining-Response-Time (SRRT) with Shortest-Remaining-Processing-Time (SRPT) because SRPT is an optimally efficient algorithm for minimizing mean response time [18]. However, the optimal efficiency of SRPT comes at the expense of starvation of large files. Thus, to study the starvation under SRRT, we compare SRRT with Processor-Sharing (PS). Since PS is the default scheduling policy in Linux, we rely to PS as a basis for defining the fairness. The fairness constraint dictates that under any algorithm no requests should finish later than under PS [12].

5.1 Experiment run

The SURGE workload generator was used to generate the server load. More than 650 thousand requests were generated in each experiment run. We used 2000 different file sizes at the server. Most files have a size less than 10KBytes. The requested file sizes ranged from 77Bytes to 3MBytes. Figure 10 shows the distribution of requests and bytes of the workload.

In our experiment, we represent the system load by the number of concurrent users, defined as the number of user's equivalents (UEs) generated by the SURGE workload generator. The web server was run under different loads (numbers of UEs). The UEs vary from 60 to 1200 for the link capacity of 100Mbps, and from 18 to 360 UEs for the link capacity

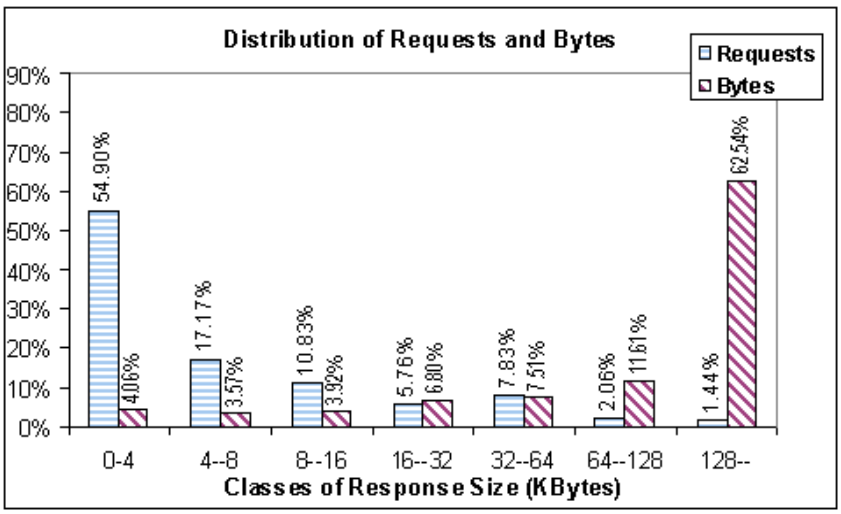


Figure 10: Distribution of the requests and bytes

of 10Mbps. In the case of a 100Mbps link, 960 UEs saturated the link. So at 960 UEs the system load is 100%. For a 10Mbps link, 120 UEs can saturate the link.

For each number of UEs, the experiment is run for 15 minutes to ensure that all jobs were completed. For each run we measure the corresponding mean response time at the client side by using the *pbvalclient* program from the SURGE package. SURGE generates an output log file that has the following format:

<client ID>, <process ID>, <session ID>, <starttime(sec)>, <starttime (usec)>, <URL>, <file size>, <endtime (sec)>, <endtime (usec)>

The *pbvalclient* program uses the log file to print out the mean response time for all n requests ($\overline{RT} = \frac{\sum_{r=1}^{n} RT(r)}{n}$).

In our experiments we assume that clients experience heterogeneous WANs. We have divided our experimental space into six WANs; where each of the six client machines represents a different WAN that shares common WAN parameters by enabling *netem* (network emulator). The WAN factors on each client machine are shown in the Table 2. We experiment with delays between 50ms and 350ms and loss rates from 0.5% to 3.0%. This range

of values was chosen to cover values reported in the Internet Traffic Report [63] where the maximum reported value for RTT is 380ms. These WAN parameters applied to incoming (ingress) packets on the network interface of client machines as shown in Figure 11.

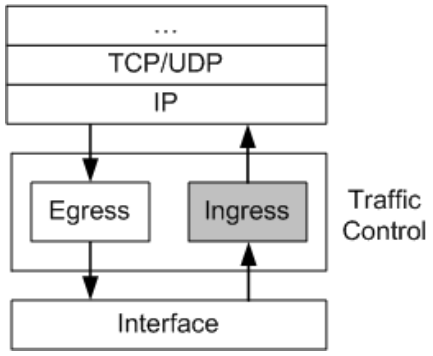


Figure 11: Linux traffic control (ingress and egress)

WANs	RTT(ms)	loss (%)
WAN1	50 ± 10	0.5
WAN2	100 ± 20	1.0
WAN3	150 ± 30	1.5
WAN4	200 ± 40	2.0
WAN5	250 ± 40	2.5
WAN6	350 ± 50	3.0

Table 2: Definition of WANs parameters

5.2 Important comments

We make some important comments about our experiment before presenting the results:

1. The network link is the bottleneck for all runs. Neither the CPU utilization nor the memory usage is the bottleneck at the server. For all runs under PS, SRPT and SRRT under 100Mbps link we monitor the CPU and RAM utilizations. The CPU utilization did not exceed 45% at the highest load (1200UEs) and the memory usage did not exceed 75%; no memory swaps occurred. For the10Mbps link, the CPU utilization is less than 7% and memory usage less than 55% for 360UEs (maximum load). Also,

at the client machines the CPU utilization did not exceed 5% and memory usage was less than 30% at the maximum load (1200 UEs) with a 100Mbps link.

2. For all experiments, the number of concurrent connections did not reach the maximum number of Apache processes (*MaxClients*) which we set to 2500.

3. The overhead for updating the priorities in SRRT and SRPT is negligible, and involving insignificant CPU overhead.

4. The same SURGE parameters were used for all the PS, SRPT and SRRT experiments.

5. In computing the mean response time, SURGE considers only those requests that completed. Thus, each experiment is run for 15 minutes to ensure that all jobs complete.

6. Although each scheduling policy has its own way of scheduling requests, they all mange to keep the link fully utilized (especially in the overload case). So the throughput is the same under PS, SRPT and SRRT.

5.3 Main results

We want to compare our algorithm, SRRT, with the existing algorithms, namely PS and SRPT. We analyze our observations from the client's point of view in terms of mean response time under the 10Mbps and 100Mbps link capacity.

The graphs in Figure 12 shows the mean response time for all WANs as a function of server load (number of UEs) for the 10Mbps and 100Mbps link capacities. Sine the workload was generated using six client machines; we just merge and sort the log files from the various clients into a single log file and then run the *pbvalclnt* program on to find the mean response time of all WANs. SRRT and SRPT show an improvement in the mean

response time over PS. Also, the SRRT shows an improvement over SRPT.

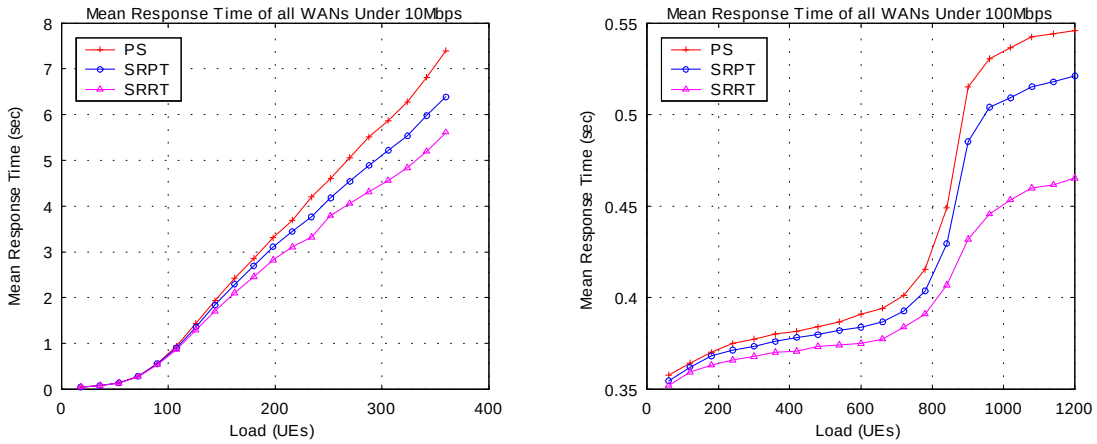


Figure 12: Mean response time of all WANs under 10Mbps and 100Mbps

Table 3 shows the improvement percentage of SRRT over SRPT and PS, in addition to the percentage improvement of SRPT over PS for the two different link capacities; 10Mbps and 100Mbbps. For each number of UEs, we concatenate all clients log files together in one file to find the mean response time at the corresponding UE. Then we calculate the average /maximum over the total range of UEs.

Improvement	**Link**	**SRRT:SRPT**	**SRRT:PS**	**SRPT:PS**
Average	10Mbps	7.5%	13.6%	6.8%
Max.		13.2%	24.2%	13.7%
Average	100Mbps	7.4%	7.1%	2.6%
Max.		11.6%	16.2%	5.8%

Table 3: Percentage improvement of SRRT and SRPT

To see in detail what is going on in each WAN, the graphs in Figure 13 and Figure 14 show the mean response time at the client side as a function of the server load (number of UEs) for WAN1 to WAN6. Each graph corresponds to a different WAN (a PC with netem) with different RTT, and loss rate. Each graph shows three curves, one for PS, one for SRPT

and one for SRRT.

5.3.1 Results for 10Mbps link capacity

From the graphs shown in Figure 13 we can observe that the graphs for the different WANs generally show almost similar behavior with respect to the mean response time. SRRT and SRPT show an improvement in the mean response time over PS. This improvement comes from the fact that the bandwidth is shared for all requests under PS. So all incomplete requests still take a fair share of the bandwidth from other requests. Hence, the mean response time of short requests (small files in SRPT, and short time in SRRT) increases. While under the SRRT and SRPT, long requests do not receive any bandwidth and short requests are completely isolated from the long requests. Therefore, completing short requests first and then long requests does not increase the mean response time by giving the chance to the small requests to complete first without competition from long requests. As a result, the PS shows a faster increase in mean response time than under SRRT and SRPT.

SRRT has the best results especially at high loads. This is likely because our approach includes, in addition to the server delay, the transmission delay which forms a large portion of the total communication delay in the Internet environment. For low loads, the three algorithms show almost similar mean response time. Since for low load the available link capacity is large enough to serve all requests, which in turn results in keeping the number of packets in the transmission queue small so that the effect of scheduling is not noticeable. However, in the low load case the RTT dominates the total communication delay so SRRT shows better behavior over SRPT in this region.

Significant improvement in performance tend to start at the point of which the link become saturated. For high load but before the link saturates (UEs <162), the improvement of SRRT over SRPT starts to become noticeable. For high load (UEs > 198), the SRRT

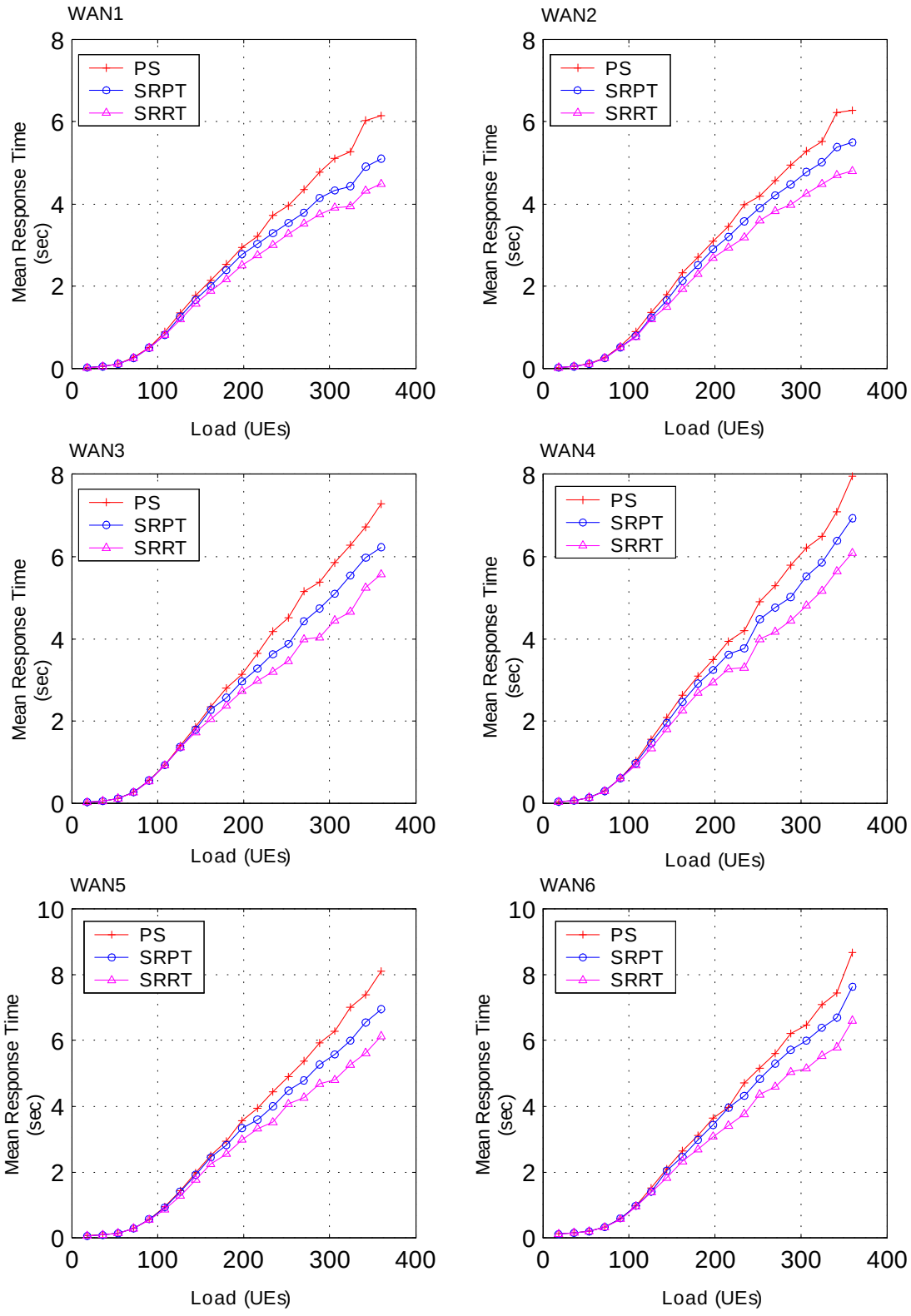


Figure 13: Mean response time as a function of workload for different WANs through 10Mbps link

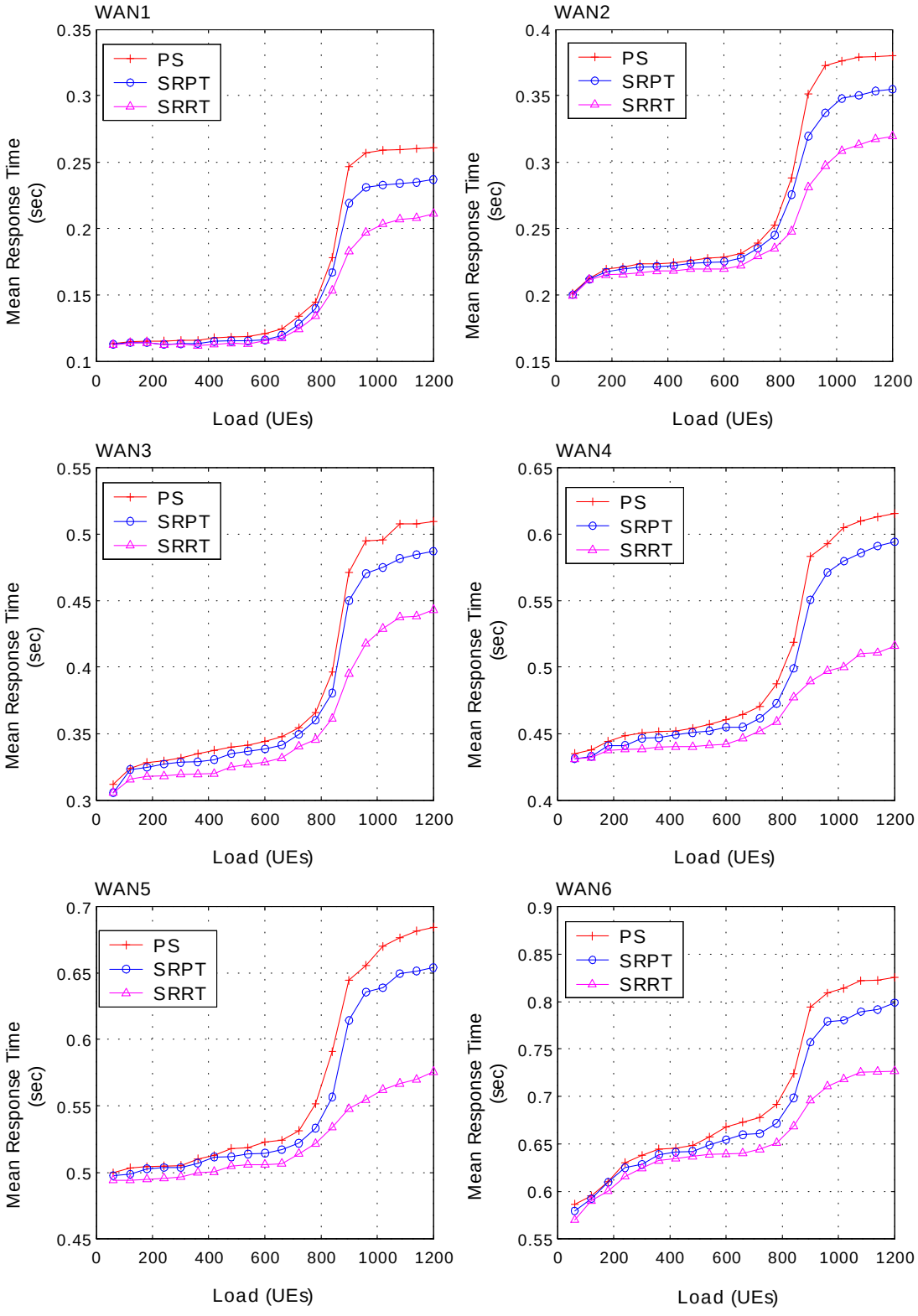


Figure 14: Mean response time as a function of workload for different WANs through 100Mbps link

shows a great improvement over the SRPT for all WANs. Tables 4 and 5 show some statistics about the improvement of SRRT over SRPT for all WANs.

WANs	**Average Improvement (%)**		
	SRRT:SRPT	**SRRT:PS**	**SRPT:PS**
WAN1	6.3	13.9	8.3
WAN2	7.0	13.9	7.5
WAN3	7.2	13.8	7.4
WAN4	7.8	13.8	6.7
WAN5	8.1	13.7	6.3
WAN6	8.2	12.6	4.9

Table 4: Average improvement percentage for 10Mbps link

WANs	**Max. Improvement (%)**		
	SRRT:SRPT	**SRRT:PS**	**SRPT:PS**
WAN1	12.0	28.4	18.7
WAN2	12.7	24.4	13.5
WAN3	15.9	25.8	14.4
WAN4	12.9	23.4	13.3
WAN5	14.4	24.9	14.5
WAN6	14.2	23.8	11.9

Table 5: Max. improvement percentage for 10Mbps link

The average percentage improvement of SRRT and SRPT over PS for 10Mbps for all WANs is shown in Figure 15. The network WAN1 has the best network conditions (Delay and loss) compared to other networks, so the requests get higher priorities under SRRT and therefore minimize the mean response time. So WAN1 has the best average improvement percentage in SRRT over PS the other WANs.

Also, we can see that bad network conditions decrease the improvement of both SRRT and SRPT scheduling techniques over PS. However, SRPT is more affected by bad network conditions than SRRT since it uses only the file size to approximate the expected response time. Server delay dominates the response time for the case of a network with no loss, and in which we ignore RTT. In contrast, under bad condition WANs (large RTT and high

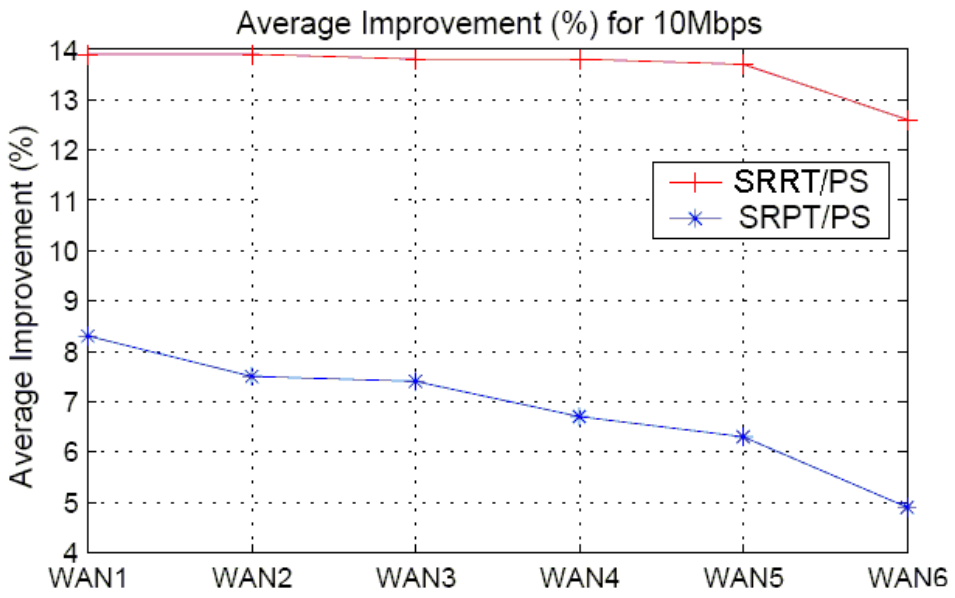


Figure 15: Average improvement of SRRT and SRPT over PS for 10Mbps link

loss rate) the transmission and retransmission delays are the dominant parts of the communication delay rather than the delay at the server. The mean response time increases as the RTT and the loss rate increase. Higher RTTs make loss recovery more expensive since the RTO (retransmission time-outs) depends on the estimated RTT. Hence, lost packets cause very long delays based on the RTT and RTO values in TCP. SRRT takes these into consideration indirectly, TCP throughput for a connection being inversely proportional to the square root of the loss [64], by decreasing the *cwnd*. When losses increase the *cwnd* decreases. Accordingly, the estimated response time in SRRT increases, so the corresponding connection receives less priority. Therefore, SRRT improvement is slightly decreased by the poor network conditions. As mentioned in [21], "While propagation delay and loss diminish the improvement of SRPT over PS, loss has a much greater effect". SRRT considers the user's network conditions by benefiting from the TCP interaction between the server and the network to take into consideration the realistic WAN factors that can dominate the mean response time.

5.3.2 Results for 100Mbps link capacity

Also for the 100Mbps link, SRRT shows the best results, especially at a high loads (Tables 6 and 7). For high load, but before the link saturates (780 < UEs < 900), the improvement of SRRT over SRPT starts to come into view. For high load (UEs > 900), the SRRT shows an improvement over the SRPT for all WANs. The average improvement in SRRT and SRPT over the PS is shown in Figure 16. The bad networks effect on both SRRT and SRPT also appears for 100Mbps.

WANs	**Average Improvement (%)**		
	SRRT:SRPT	**SRRT:PS**	**SRPT:PS**
WAN1	5.2	9.6	4.9
WAN2	5.0	8.1	3.4
WAN3	5.0	7.3	2.5
WAN4	5.2	7.2	2.3
WAN5	4.9	6.9	2.2
WAN6	3.8	5.9	2.2

Table 6: Average improvement percentage for 100Mbps link

WANs	**Max. Improvement (%)**		
	SRRT:SRPT	**SRRT:PS**	**SRPT:PS**
WAN1	16.6	25.8	11.1
WAN2	12.0	20.3	9.7
WAN3	12.3	16.2	5.1
WAN4	13.8	17.4	5.6
WAN5	12.8	16.4	5.8
WAN6	9.1	12.4	4.7

Table 7: Max. improvement percentage for 100Mbps link

From the above we can see that the SRRT always works well under all network conditions, no matter which component of the total communication time dominates since it takes into consideration the RTT and the TCP throughput rate, in addition to the file size, in

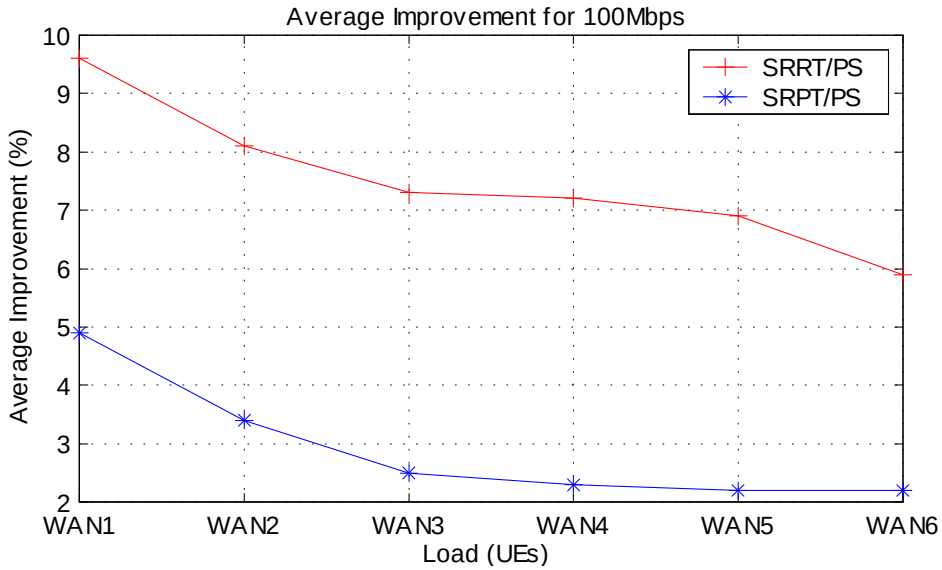


Figure 16: Average improvement of SRRT and SRPT over PS for 100Mbps link

approximating the response time.

5.3.3 Starvation analysis

To see whether the improvement in mean response time comes at the expense of starvation for long requests, we look to the response time for each individual request under SRPT and SRRT scheduling algorithms. To quantify the starvation, we use the starvation stretch metric which is introduced in [12]. Starvation stretch $S_x(r)$ of request r under algorithm X is the ratio of response time $RT_x(r)$ under algorithm X to response time $RT_{ps}(r)$ under PS:

$$S_x(r) = \frac{RT_x(r)}{RT_{ps}(r)}$$

The starvation occurs under the algorithm X if the $S_x(r) > 1$.

5.3.3.1 Starvation under SRPT

Under SRPT, we found that 2.3% of the response has starvation stretch greater than 1 under the 100Mbps link capacity, and the largest file (3119822Bytes) has a starvation stretch of

2.1. Under the 10Mps capacity, 2.6% of the response times starved. The largest file has a starvation stretch of 2.4.

Mor Harchol et al. [13] show that for job size distributions with the heavy tailed property, more than 98% of the requests have a considerable improvement in mean response time under SRPT compared with PS.

Figure 10 shows the distribution of requests and bytes of our workload. About 82.9% of the requests are for files smaller than 16KB and they represent 11.55% of the total requested bytes. If the server gives preference to small files (less than 16kB), 88.45% of the server resources will be in use for remaining 17.09% of large files (greater than 16kB). Only 1.44% of the large requests (greater than 128kB) form 62.54% of the requested bytes. Considering this property of the workload, most of the server resources will be dedicated to large requests. So the large request will be negligibly penalized under SRPT and SRRT as compared with the PS scheduling.

5.3.3.2 Starvation of SRRT

SRRT is based on SRPT, but it differs in the way that estimates the response time. The SRRT shows better performance than SRPT since it has more information about the response time. For SRRT only 1.5% of the long response starved under the 100Mbps link. The longest response has a starvation stretch 1.7. Under the 10Mbps, 2.4% of the requests starved. The longest response has a starvation stretch 2.2.

5.3.4 Sources of performance of SRRT over SRPT

SRPT assumes that the response time of a request is strongly correlated with the size of the requested file. Dong Lu [25] shows that the correlation between the response time and the file size is unwarranted. This low correlation between the file size and the response time

comes from the nature of large networks that show path diversity to the clients; every path will likely have a different RTT and different bandwidth of the path. So the response time is not strongly correlated with the file size. Therefore, SRPT is strongly affected by this weak correlation between the file size and the response time [25].

Although the transfer time is likely to be dominated by the transmit delay and RTT of the path, there are other possible delays, due to packet loss and congestion. Thus, the complexity and the diversity of the WAN environment suggest that the response time of requests may not be proportional to the size of the file it serves. This led us to find better estimators for response time using SRRT which make scheduling policy perform better. Our approach tries to gain a more exact knowledge of the response times:obtained from TCP to improve web server performance.

Our approach adds an additional overhead compared to SRPT since it needs to call *getsockopt()* to get the RTT and the *cwnd*. However, this additional overhead is not critical under the assumption that the CPU is not the bottleneck. We found about a 1% increase in the CPU utilization under SRRT over the SRPT.

CHAPTER VI

CONCLUSION AND FUTURE WORK

The client-server architecture is mainly used in the Internet environment which has high diversity in bandwidth, propagation delay and packet loss rate. Therefore, the SRPT assumption that file size is a good approximation for the total response time for a static request on a web server is not accurate and the performance of SRPT degrades dramatically in a WAN environment. For that reason, we proposed SRRT to better estimate the response time by getting useful TCP information, which is available at web server about the connection, in addition to the file size, without producing additional traffic. The SRRT uses the RTT and the congestion window size and the file size to approximate the response time. The request with shortest SRRT receives the highest priority.

We proposed, implemented and evaluated a new scheduling policy for web servers. The proposed policy, SRRT, improves the client-perceived response time, which is the main factor determining the request scheduling policy, in comparison to the default Linux scheduling (PS) and the SRPT scheduling policies. The SRRT performs better than SRPT and PS at high and moderate uplink load and especially under overload condition. The performance improvement is achieved under different uplink capacities, for a variable range of network parameters (RTT's and loss rate). This improvement does not unduly penalized the long requests and without loss in byte throughput.

The implementation of SRRT was done on an Apache web server running Linux to prioritize the order that the socket buffers are drained within the kernel. The priority

of the requests is determined based on the priority array values we have coded in the Apache source code. The choice of these values is based on the experiment trials. After experimenting with different values, we found that the values adopted gave us good results. But we do not claim that this choice is optimal. Also, it is better to make these values configurated by the Apache configuration file to enable the administrator to change them as needed.

Another improvement on SRRT may be done by trying to take other factors that may affect the response time like queue delay approximation and the TCP connection loss rate. In evaluating of SRRT, we represent the WAN environment with different RTTs and loss rates by using a network emulator. We have not been able to evaluate all real world application parameters. To check the validity of this algorithm, it is better to test it on a real web server. Also, it is good to evaluate the SRRT algorithm analytically to examine the validity of the experimental results.

The SRRT is applied to static web requests with heavy tailed distribution file sizes. Future work can be enhancing it to also schedule dynamic requests where the approximation of the response time is not as easy as for static requests. Also, this work may extend to other operating systems and for other file size distributions.

We believe that SRRT scheduling will continue to be applicable in the future, although better link speeds become available and the bandwidth cost decreases. Due to financial constrains, many users will not upgrade their connectivity conditions. Also, the variance in network distance and environment will persist and diversity in delay will be continue to exist.

REFERENCES

[1] A. M. Odlyzko. “Internet traffic growth: Sources and implications.” *Optical Transmission Systems and Equipment for WDM Networking II*, vol. 5247, pp. 1-15, 2003.

[2] J. Nielsen. "One billion Internet users." Internet: http://www.useit.com/alertbox/internet_growth.html, Dec. 19, 2005.

[3] :Internet World stats: usage and population statistics." Internet:http://www.internetworldstats.com/stats.htm, Jun. 10, 2007.

[4] D.Sullivan. "Searches per day." Internet: http://searchenginewatch.com/reports/article.php/2156461, Apr. 20, 2006.

[5] A. B. King. "Speed up your site: web site optimization." Internet: http://www.websiteoptimization.com/speed, Jan. 17, 2003.

[6] J. Nielsen. "The need for speed." Internet: http://www.useit.com/alertbox/9703a.html, Mar. 1, 1997.

[7] D. Kegel. "The C10K problem." Internet: http://www.kegel.com/c10k.html, Sep. 02, 2006.

[8] M. Crovella, R. Frangioso, and M. Harchol-Balter. "Connection scheduling in web servers." *USENIX Symposium on Internet Technologies and Systems*, Oct. 1999.

[9] M. Harchol-Balter, M. Crovella, S. Park. "The case for SRPT scheduling in web servers." Tech. Rep. MIT-LCS-TR-767, Carnegie Mellon School of Computer Science, Oct. 1998.

[10] M. Rawat and A. Kshemkayani. "SWIFT: Scheduling in web servers for fast response time." *Second IEEE International Symposium on Network Computing and Applications*, Apr. 2003.

[11] M. Harchol-Balter, N. Bansal, B. Schroeder, and M. Agrawal. "Implementation of SRPT scheduling in web servers." Tech. Rep. CMU-CS-00-170, Carnegie Mellon School of Computer Science, Oct. 2000.

[12] C. Jechlitschek and S. Gorinsky. “Fair Efficiency, or Low Average Delay without Starvation.” Tech. Rep. WUCSE-2007-16, Washington University in St. Louis, US, Feb. 28, 2007.

[13] N.Bansal and M. Harchol-Balter. "Analysis of SRPT scheduling: investigating unfairness." *ACM SIGMETRICS Performance Evaluation Review*, vol. 29, no. 1, pp. 279-290, June 2001.

[14] P. Barford and M. E. Crovella." Critical path analysis of TCP transactions." *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 2, pp. 80-102, Apr. 2001.

[15] S. Manley and M. Seltzer. "Web facts and fantasy." *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, 1997.

[16] C. D. Murta and T. P. Corlassoli. "Fastest connection first: A new scheduling policy for web servers." In the 18th International Teletra#c Congress (ITC-18), Sep. 2003.

[17] L.E. Schrage and L. W. Miller. "The queue M/G/1 with the shortest remaining processing time discipline." *Operations Research*, vol.14, no. 4, pp. 670-684, Jul. 1966.

[18] L. E. Schrage. "A proof of the optimality of the shortest remaining processing time discipline." *Operations Research*, vol.16, no. 3, pp. 678-690, 1968.

[19] D. R. Smith. "A new proof of the optimality of the shortest remaining processing time discipline". *Operations Research*, vol. 26, no. 1, pp. 197-199. Jan.1976.

[20] C. Goerg. "Evaluation of the optimal SRPT strategy with overhead." *IEEE Transactions on Communications*, vol. 34, pp. 338-344. Apr. 1986.

[21] M. Harchol-Balter, B. Schroeder, M. Agrawal, N. Bansal. "Size-based scheduling to improve web performance." *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 2, pp. 207-233, May 2003.

[22] B. Schroeder and M. Harchol-Balter. "Web servers under overload: How scheduling can help." *ACM Transactions on Internet Technology (TOIT)*, vol. 6, no. 1, pp. 20-52, Feb. 2006.

[23] M.Gong and C.Williamson. "Quantifying the properties of SRPT scheduling". *Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, pp. 126-135, Oct. 2003.

[24] E.. J. Friedman and S. G. Henderson. "Fairness and efficiency in web server protocols." *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 229-237, 2003.

[25] D. Lu, H. Sheng, P. A. Dinda. "Effects and implications of file size/service time correlation on web server scheduling policies." *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 258-267, Sep. 2005.

[26] N. Bansal. "On the average sojourn time under M/M/1 SRPT." *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 2, pp. 34-35, September 2003.

[27] N.Bansal and D. Gamarnik. "Handling load with less stress." *Queueing Systems*, vol. 54 , no. 1, pp. 45-54, Sep. 2006.

[28] M. Harchol-Balter, K. Sigman, A.Wierman. "Asymptotic convergence of scheduling policies with respect to slowdown." *Performance Evaluation*, vol. 49, no.1-4, pp. 241–256, Sep. 2002.

[29] J. Guitart, V. Beltran, D. Carrera, J. Torres, E. Ayguadé. "Session-based adaptive overload control for secure dynamic web application.″ International Conference on Parallel Processing (ICPP2005), vol. 00, pp. 341-349, Jun. 2005.

[30] X. Chen, H. Chen, P.Mohapatra. ACES: "An efficient admission control scheme for QoS-aware web servers." *Computer Communications*, vol. 26, no.14, pp. 1581-1593, Sep. 2003.

[31] M.Welsh and D.Culler. *"Adaptive overload control for busy Internet servers." 4th Symposium on Internet Technologies and Systems*, Seattle, WA, USA. Mar. 2003.

[32] S. Elnikety, E. Nahum, J. Tracey, and W Zwaenepoel. "A method for transparent admission control and request scheduling in e-commerce web sites." 13th International Conference on World Wide Web (WWW 2004), pp. 276-286, New York, NY, USA. May 2004.

[33] B.Krishnamurthy, Craig E. Wills. "Improving web performance by client characterization driven server adaptation." 11th international conference on World Wide Web (WWW2002), pp. 305-316, Honolulu, Hawaii, USA, May 2002.

[34] P. Barford and M.Crovella. "Generating representative web workloads for network and server performance evaluation." *ACM joint international conference on Measurement and modeling of computer systems*, pp. 151-160, Jun. 1998.

[35] Standard Performance Evaluation Corporation." SPECweb96 and SPECweb99." Internet: http://www.spec.org, May 23, 2007.

[36] hp research labs. "Httperf: HTTP benchmarking utility." Internet: http://www.hpl.hp.com/research/linux/httperf, Jun. 10, 2005.

[37] "Differentiated Services on Linux." Internet: http://diffserv.sourceforge.net, May 29, 2001.

[38] W. Almesberger. "Linux traffic control - next generation." Internet: http://tcng.sourceforge.net, Oct. 2002.

[39] M. A. Brown." Traffic control HOWTO." Internet: http://linux-ip.net/articles/Traffic-Control-HOWTO/classless-qdiscs.html, Oct. 2006.

[40] T. Graf, G. Maxwell, R. Mook, M. Oosterhout, P. Schroeder, J. Spaans and P. Larroy. "Linux advanced routing & traffic control HOWTO." Internet: http://lartc.org/howto.

[41] L. Balliache. "Differentiated service on Linux HOWTO." Internet: http://www.opalsoft.net/qos/DS-23.htm, Aug. 2003.

[42] National Institute of Standard and Technology. "The NIST Net network emulator." Internet: http://snad.ncsl.nist.gov/nistnet, Mar. 14 , 2001.

[43] V. N. Padmanabhan and K. Sripanidkulchai. "The Case for Cooperative Networking", *IPTPS*,vol. 2429, Mar. 2002.

[44] L.Rizzo. "Dummynet network emulator." Internet: http://info.iet.unipi.it/~luigi/ip_dummynet.

[45] "Network Emulation (Netem)." Internet: http://linux-net.osdl.org/index.php/Netem, 8 Apr. 8, 2007.

[46] B. Curry, T. Writer, G. V. Reilly, IIS Performance Team, and H. Kaldestad, ITG Internet Hosting Services." The art and science of web server tuning with Internet information services 5.0." Internet: http://www.microsoft.com/technet/prodtechnol/windows2000serv/technologies/iis/maintain/optimize/iis5tune.mspx, Mar. 9, 2001 [Jun. 10, 2007].

[47] C. MacC´arthaigh. "Scaling Apache 2.x beyond 20,000 concurrent downloads." Internet: http://www.stdlib.net/~colmmacc/Apachecon-EU2005/scaling-apache-handout.pdf, Jul. 21, 2005.

[48] Lawrence Berkeley National Laboratory." TCP Tuning Guide." Internet: http://dsd.lbl.gov/TCP-tuning/linux.html, Feb 13, 2006.

[49] RFC 793 – Transmission Control Protocol, September 1981.

[50] RFC 1122 - Requirements for Internet Hosts - Communication Layers, October 1989.

[51] RFC 1323 - TCP Extensions for High Performance, May 1992.

[52] RFC 2018 - TCP Selective Acknowledgment Options, October 1996.

[53] RFC 2581 – TCP Congestion Control, April 1999.

[54] RFC 2988 - Computing TCP's Retransmission Timer, November 2000.

[55] RFC 3168 – The Addition of Explicit Congestion Notification (ECN) to IP, September 2001.

[56] RFC 3390 – Increasing TCP's Initial Window, October 2002.

[57] RFC 3742 – Limited Slow-Start for TCP with Large Congestion Windows, March 2004.

[58] The NewReno Modification to TCP's Fast Recovery Algorithm, April 2004.

[59] P. Karn and C. Partridge. "Improving round-trip time estimates in reliable transport protocols." *ACM SIGCOMM Computer Communication Review,* vol. 25, no. 1, pp. 66-74, Jan. 1995.

[60] V. Jacobson. "Congestion avoidance and Control." *ACM SIGCOMM Computer Communication Review*, vol. 25, no.1, pp. 157-187, Jan. 1995.

[61] D. Comer. *Internetworking with TCP/IP*, vol.1 (5^{th} edition), pp. 212-213, Jan. 2005.

[62] http://www.ssfnet.org/Exchange/tcp/tcpTutorialNotes.html and http://cities.lk.net/tcp.html.

[63] "Internet Traffic Report. 2007." Internet: http://www.internettrafficreport.com

[64] J. Padhye, V. Firoiu, D. F. Towsley, , and J. F. Kurose. "Modeling tcp reno performance: A simple model and its empirical validation." I*EEE/ACM Transactions on Networking (TON)*, vol. 8, no. 2, pp133-145, Apr. 2000.

APPENDIX A

SHORTEST TOTAL ESTIMATED RESPONSE TIME (STERT)

In the STERT, the priorities are given to the requests based on the shortest total estimated response time (STERT) and not only on the remaining estimated response time (RRT). The total estimated response time (TERT) is the sum of the elapsed time (ET) between the start of the transmission for a request and the remaining response time (RRT). To determine the elapsed time (ET), we need to know the start time for each request and the current time. If we know these values, the elapsed time will be the current time (CT) minus the request start time (RST).

$$ET = CT - RST$$

In Apache code, the request object (r->$request\_time$) stores the request start time, so we get it without an additional system call. To obtain the current time, we use the *gettimeofday()* function.

Thus, the total estimated response time (TERT) is

$$TERT = RRT + ET$$

ET is exactly known, while RRT is an estimate value. The RRT can be approximated as follows:

$$RRT = RTT + \frac{RFS}{R}$$

where RFS is the remaining file size and R is the transfer rate for a connection

The estimated current maximum transfer rate (R_C) for a connection can be approximated by:

$$R_C = \left(\frac{cwnd}{RTT}\right)^1$$

R_C is based on the current estimated RTT value and the current *cwnd* size from the TCP_INFO.

The past transfer rate (R_P) for a connection can be calculated by:

$$R_P = \frac{FS - RFS}{ET}, \text{ where FS is the total file size}$$

We can estimate the transfer rate for a connection as follow:

$$R = \alpha R_P + (1 - \alpha) R_C, \text{ where } \alpha = 0.9$$

Finally, we can write the total estimated response time (TERT) as follows:

$$TERT = (RTT + \frac{RFS}{R}) + ET$$

The first term $(RTT + \frac{RFS}{R})$ is an estimate value, while ET is an exact value.

As shown in Figure 17 the shortest total estimated response time (STERT) shows a better improvement in mean response time over SRRT for all WANs as a function of server Loads (number of UEs) under 10Mbps and 100Mbps link capacities. While SRRT shows the best results over STERT, SRRPT, and PS. Thus, the scheduling based on the remaining response time is better than the scheduling based on the total response time.

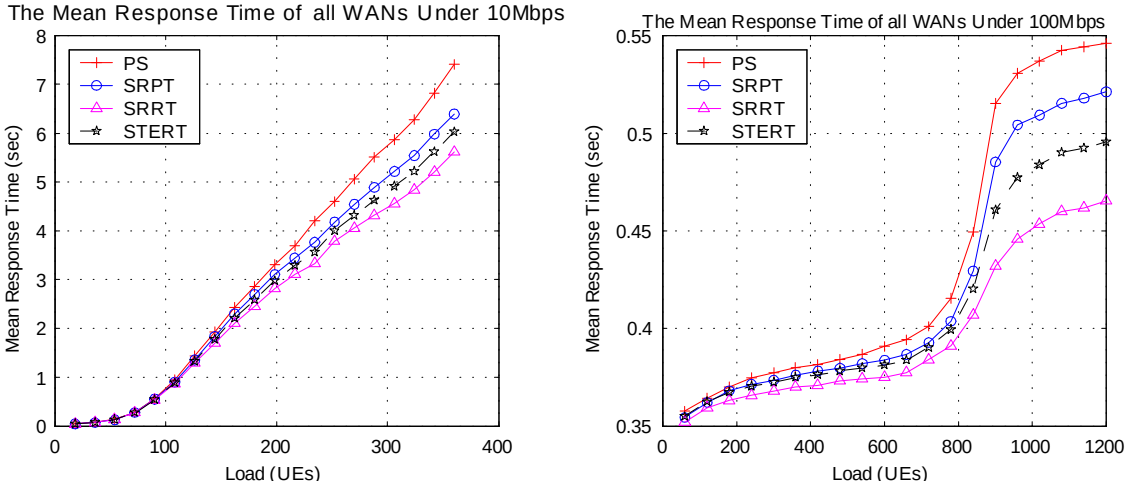


Figure 17: Mean response time of all WANs under 10Mbps and 100Mbps

APPENDIX B

APACHE SOURCE CODE MODIFICATIONS

We modified only two files; *http_protocol.c* and *http_core.c*. The diffs for these files were shown below:

B.1 The modifications to implement SRRT

B.1.1 Changes to http_protocol.c file

```
105a106,143
> /* Change to implement SRRT: */
> /* Here we add our function to set the priority of a file corresponding to */
> /* the remaining data left to send for this file */
>
> #include <netinet/tcp.h>
> void sctf_set_sock_prio(int fd, unsigned int remaining_len) {
>
> //modification ...
> struct tcp_info tcpinfo;
> socklen_t leng;
>
> leng = sizeof(tcpinfo);
> getsockopt(fd, IPPROTO_TCP, TCP_INFO, &tcpinfo, &leng);
>
> unsigned int rtt_ms = tcpinfo.tcpi_rtt/1000;
> unsigned int mss = tcpinfo.tcpi_advmss;
> unsigned int cwnd_byte = (tcpinfo.tcpi_snd_cwnd)*mss;
>
> unsigned int comm_time;
> comm_time = rtt_ms + (remaining_len/cwnd_byte)*(rtt_ms);
>
> unsigned int prios[7] = { 0, 100, 500, 1000, 10000, 100000, 10000000};
> unsigned int prio;
> int i = 0;
>
> while((comm_time > prios[i]) && i<=6) {
> ++i;
> }
> prio=i;
```

```
>
> setsockopt(fd, SOL_SOCKET, SO_PRIORITY, &prio, sizeof(prio));
> }
>
> /* End Change */
>
2157a2196,2203
>
> /* Change to implement SRRT: */
> /* update the priority of the socket */
>
> sctf_set_sock_prio(r->connection->client->fd, r->finfo.st_size-total_bytes_sent);
>
> /* End change */
>
2326c2372
< if (length - offset > MMAP_SEGMENT_SIZE) {
---
> if (length - offset > MMAP_SEGMENT_SIZE) {
2333a2380,2387
>
> /* Change to implement SRRT: */
> /* update the priority of the socket */
>
> sctf_set_sock_prio(r->connection->client->fd, length-offset);
>
> /* End Change */
```

B.1.2 Changes to http_core.c file

```
3167a3168,3175
>
> /* Change to implement SRRT: */
> /* update the priority of the socket */
>
> sctf_set_sock_prio(r->connection->client->fd, r->finfo.st_size);
>
> /* End Change */
```

B.2 The modifications to implement SRPT

B.2.1 Changes to http_protocol.c file

```
105a106,126
> /* Change to implement SRPT: */
> /* Here we add our function to set the priority of a file corresponding to */
> /* the remaining data left to send for this file */
>
```

```
> unsigned int prios[7] = { 0, 1000, 10000, 15000, 30000, 1000000, 10000000};
>
> void srpt_set_sock_prio(int fd, unsigned int remaining_len) {
> unsigned int prio;
> int i = 0;
>
> while((remaining_len > prios[i]) && i<=6) {
> ++i;
> }
> prio=i;
>
> setsockopt(fd, SOL_SOCKET, SO_PRIORITY, &prio, sizeof(prio));
> }
>
> /* End Change */
>
2157a2179,2187
>
> /* Change to implement SRPT: */
> /* update the priority of the socket */
>
> srpt_set_sock_prio(r->connection->client->fd, r->finfo.st_size-total_bytes_sent);
>
> /* End change */
>
2326c2356
< if (length - offset > MMAP_SEGMENT_SIZE) {
---
> if (length - offset > MMAP_SEGMENT_SIZE) {
2333a2364,2371
>
> /* Change to implement SRPT: */
> /* update the priority of the socket */
>
> srpt_set_sock_prio(r->connection->client->fd, length-offset);
>
> /* End Change */
```

B.2.2 Changes to http_core.c file

```
3167a3168,3175
>
> /* Change to implement SRPT: */
> /* update the priority of the socket */
>
> srpt_set_sock_prio(r->connection->client->fd, r->finfo.st_size);
>
> /* End Change */
```