Size-based Scheduling to Improve Web Performance

MOR HARCHOL-BALTER, BIANCA SCHROEDER, NIKHIL BANSAL, and
MUKESH AGRAWAL
Carnegie Mellon University

Is it possible to reduce the expected response time of *every* request at a web server, simply by changing the order in which we schedule the requests? That is the question we ask in this paper.

This paper proposes a method for improving the performance of web servers servicing static HTTP requests. The idea is to give preference to requests for small files or requests with short remaining file size, in accordance with the SRPT (Shortest Remaining Processing Time) scheduling policy.

The implementation is at the kernel level and involves controlling the order in which socket buffers are drained into the network. Experiments are executed both in a LAN and a WAN environment. We use the Linux operating system and the Apache and Flash web servers.

Results indicate that SRPT-based scheduling of connections yields significant reductions in delay at the web server. These result in a substantial reduction in mean response time and mean slowdown for both the LAN and WAN environments. Significantly, and counter to intuition, the *requests for large files* are only negligibly penalized or not at all penalized as a result of SRPT-based scheduling.

Categories and Subject Descriptors: C.5.5 [**Computer System Implementation**]: Servers; D.4.1 [**Operating Systems**]: Process Management—*scheduling*; D.4.8 [**Operating Systems**]: Performance—*Queueing theory*

General Terms: Performance, Algorithms, Design, Experimentation, Measurement

Additional Key Words and Phrases: Conservation law, networking, scheduling, SRPT, SJF, system performance and design, web servers

1. INTRODUCTION

A client accessing a busy web server can expect a long wait. This paper considers how we might reduce this wait for the case of *static* requests, of the form "Get me a file." Evidence suggests that while the number of dynamic requests are growing, the majority of the requests at most web servers are still *static* [Manley and Seltzer 1997], [Krishnamurthy and Rexford 2001] (p.8), [Feldmann], [Arlitt et al. 1999]. Serving static requests *quickly* is the focus of many companies *e.g.*, Akamai Technologies, and much ongoing research.

This research is funded by Cisco Systems and Spinnaker Networks via a grant from the Pittsburgh Digital Greenhouse 00-1, by NSF-ITR 99-167 ANI-0081396, and by NSF Career Grant CCR-0133077.
Author's address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213; email: harchol@cs.cmu.edu.

In this paper we are concerned with *response time* which is defined to be the time from when the client sends out the SYN-packet requesting to open a connection until the client receives the last byte of the file requested.

Our idea is simple. Traditionally, requests at a web server are time-shared: the web server proportions its resources fairly among those requests ready to receive service. We call this scheduling policy **FAIR** scheduling. We propose, instead, *unfair scheduling*, in which priority is given to requests for short files, or those requests with short remaining file size, in accordance with the well-known scheduling algorithm preemptive Shortest-Remaining-Processing-Time-first (**SRPT**). It is well-known from queueing theory that SRPT scheduling minimizes queueing time, [Schrage and Miller 1966]. Allowing short requests to preempt long requests is desirable because forcing long requests to wait behind short requests results in much lower mean response time than the situation where short requests must wait behind long requests. Our expectation is that using SRPT scheduling of requests at the server will reduce the queueing time at the server, and therefore the total response time.

Despite the obvious advantages of SRPT scheduling with respect to mean response time, applications have shied away from using this policy for two reasons: First SRPT requires knowing the time to service the request. Our experiments show that the time to service a request is well-approximated by the size of the file requested, which is well-known to the server. We found a linear relationship between the service time of the request and its file size, modulo a small overhead. Second, there is the fear that SRPT "starves" requests for large files [Bender et al. 1998], [Stallings 2001] (p. 410), [Silberschatz et al. 2002] (p. 162). A primary goal of this paper is to investigate whether this fear is valid in the case of web servers serving typical web workloads.

It is not immediately clear what SRPT means in the context of a web server. A web server is not a single-resource system. To understand *which* of the web server's resources need to be scheduled, we need to understand which resource in a web server experiences high load first, i.e., which is the *bottleneck* resource. The three contenders are: the CPU; the disk to memory bandwidth; and the server's limited fraction of its ISP's bandwidth. On a site consisting primarily of *static content*, a common performance bottleneck is the limited bandwidth which the server has bought from its ISP [Microsoft 2001; Cockcroft 1996; Maggs 2001]. Even a fairly modest server can completely saturate a T3 connection or 100Mbps Fast Ethernet connection. Also, buying more bandwidth from the ISP is typically relatively more costly than upgrading other system components like memory or CPU.

In this paper, we model the limited bandwidth that the server has purchased from its ISP by placing a limitation on the server's uplink, as shown in Figure 1. In all our experiments (using both a 10Mbps and 100 Mbps uplink, and 256 MB of RAM, and running various trace-based workloads) the bandwidth on the server's uplink is always the bottleneck resource. The system load is therefore defined in terms of the load on the server's uplink, which we refer to as the **uplink load**. For example, if the web server has a 100 Mbps uplink and the average amount of data requested by the clients is 80 Mbps, then the uplink load is 0.8. Although in this paper we assume that the bottleneck resource is the limited bandwidth that the server has purchased from its ISP, the main ideas can also be adapted for alternative bottleneck resources.

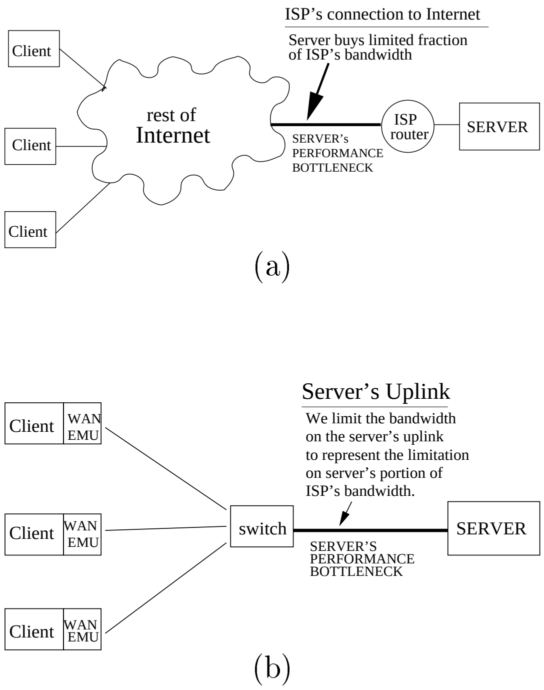


Fig. 1. *(a) Server's bottleneck is the limited fraction of bandwidth that it has purchased from its ISP. (b) How our implementation setup models this bottleneck by limiting the server's uplink bandwidth.*

The focus in the rest of the paper is on how to schedule the server's uplink bandwidth, and the performance effects of this scheduling. To schedule the server's uplink bandwidth, we need to apply the SRPT algorithm at the level of the network. Our approach is to control the order in which the server's socket buffers are drained. Recall that for each (non-persistent) request a connection is established between the client and the web server. Corresponding to each connection, there is a socket buffer on the web server end into which the web server writes the contents of the requested file. Traditionally, the different socket buffers are drained in Round-Robin Order, with equal turns being given to each eligible connection, where an eligible connection is one that has packets waiting to be sent and for which TCP congestion control allows packets to be sent. Thus each eligible connection receives a fair share of the bandwidth of the server's uplink. We instead propose to give priority to those sockets corresponding to connections requesting small files or where the *remaining data* required by the request is small. Throughout, we use the Linux OS.

The goal of this paper is to compare FAIR scheduling with SRPT scheduling. These are defined as follows:

FAIR scheduling: This uses standard Linux (fair-share draining of socket buffers) with an unmodified web server.

SRPT scheduling: This uses modified Linux (SRPT-based draining of socket buffers) with the web server modified only to update socket priorities.

We experiment with two different web servers: the common Apache server [Apache 2001], and the Flash web server [Pai et al. 1999], which is known for speed. Since results are quite similar, we primarily show here only the results for the case of Apache, and leave the Flash results for the associated technical report [HarcholBalter et al. 2000]. Our clients make requests according to a web trace, which specifies both the time the request is made and the size of the file requested. Experiments are also repeated using requests generated by a web workload generator.

Experiments are executed first in a LAN, so as to isolate the reduction in queueing time at the server. Response time in a LAN is dominated by queueing delay at the server and TCP effects. Experiments are next repeated in a WAN environment. The WAN allows us to incorporate the effects of propagation delay, network loss, and congestion in understanding more fully the client experience. WAN experiments are executed both using a WAN emulator and by using geographically-dispersed client machines.

Synopsis of results obtained for a LAN:

—SRPT-based scheduling decreases mean response time in a LAN by a factor of 3 – 8 for uplink loads greater than 0.5.

—SRPT-based scheduling helps requests for small files a lot, while negligibly penalizing requests for large files. Under an uplink load of 0.8, 80% of the requests improve by a factor of 10 under SRPT-based scheduling. Only the request for the largest file suffers an increase in mean response time under SRPT-based scheduling (by a factor of only 1.2).

—There is no negative effect on network throughput or CPU utilization from using SRPT as compared with FAIR.

Synopsis of results obtained for a WAN:

—While propagation delay and loss diminish the improvement of SRPT over FAIR, loss has a much greater effect.

—For an RTT of 100ms, under an uplink load of 0.9, SRPT's improvement over FAIR is still a factor of 2.

—Network loss diminishes the improvement of SRPT over FAIR further. Under high network loss (10%), SRPT's improvement over FAIR is only 25% under an uplink load of 0.9.

—Unfairness to requests for large files remains negligible or non-existent under WAN conditions.

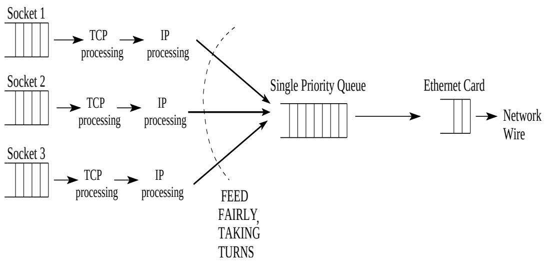
Section 2 describes our implementation of SRPT scheduling. Section 3 describes the LAN experimental setup and the LAN results. Section 4 describes the WAN experimental setup and the WAN results. Section 5 provides an in depth look at *why* SRPT scheduling improves over FAIR scheduling. Section 6 describes previous work. Finally in Section 7, we elaborate on broader applications of SRPT-based scheduling, including its application to other resources, and to non-static requests. We also discuss SRPT applied to web server farms and Internet routers.

2. IMPLEMENTATION OF SRPT

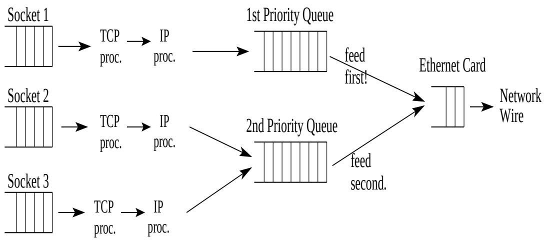
In Section 2.1 we explain how socket draining works in standard Linux, and we describe how to achieve priority queueing in Linux (versions 2.2 and above). Section 2.2 describes the implementation end at the web server and also deals with the algorithmic issues such as how to choose good *priority classes* and the setting and updating of priorities. Furthermore we consider the problem that for small file requests, a large portion of the time to service the request is spent *before* the size of the requested file is even known, and we find a solution for this problem.

2.1 Achieving priority queueing in Linux

Figure 2(a) shows data flow in standard Linux.



(a) Standard Linux – FAIR



(b) Modified Linux – SRPT

Fig. 2. *(a) Data flow in standard Linux. The important thing to observe is that there is a* single *priority queue into which all ready connections drain fairly. (b) Linux with priority queueing. There are several priority queues, and queue i is serviced only if all of queues* 0 *through $i-1$ are empty.*

There is a socket buffer corresponding to each connection. Data streaming into each socket buffer is encapsulated into packets which obtain TCP headers and IP headers. Throughout this processing, the packet streams corresponding to each connection are kept separate. Finally, there is a *single*[1] "priority queue" (*transmit queue*), into which *all* streams feed. All eligible streams (eligible via TCP congestion

[1]The queue actually consists of 3 priority queues, a.k.a. bands. By default, however, all packets are queued to the same band.

control) take *equal* turns draining into the priority queue. Although the Linux kernel does not explicitly enforce fairness, we find that under conditions where clients are otherwise equal, TCP governs the flows so that they share fairly on short time scales. This single "priority queue," can get as long as 100 packets. Packets leaving this queue drain into a short Ethernet card queue and out to the network.

To implement SRPT we need more priority levels. To do this, we first build the Linux kernel with support for the user/kernel Netlink Socket, QOS and Fair Queueing, and the Prio Pseudoscheduler. Then we use the `tc` [Almesberger 1999] user space tool to switch the Ethernet card queue from the default 3-band queue to the 16-band prio queue. Further information about the support for differentiated services and various queueing policies in Linux can be found in [Radhakrishnan 1999; Almesberger 1999; Almesberger et al. 1999].

Figure 2(b) shows the flow of data in Linux after the above modification: The processing is the same until the packets reach the priority queue. Instead of a single priority queue (transmit queue), there are 16 priority queues. These are called bands and they range in number from 0 to 15, where band 15 has lowest priority and band 0 has highest priority. All the connections of priority i feed fairly into the ith priority queue. The priority queues then feed in a prioritized fashion into the Ethernet Card queue. Priority queue i is only allowed to flow if priority queues 0 through $i-1$ are all empty.

A note on experimenting with the above implementation of priority queueing: Consider an experiment where each connection is assigned to one of two priorities. We have found that when the number of simultaneous connections is very low, the bandwidth is not actually split such that the first priority connections get 100% of the bandwidth and the second priority connections get 0% of the bandwidth. The reason is that with very few connections, the first priority connections are unable to fully utilize the link, and thus the second priority connections get a turn to run. However, when the number of simultaneous connections is higher (e.g., above 10), this is not a problem, and the first priority connections get 100% of the bandwidth. In all the experiments in this paper, we have hundreds of simultaneous connections and the above implementation of priority queueing works perfectly.

2.2 Modifications to web server and algorithmic issues in approximating SRPT

The modified Linux kernel provides a mechanism for prioritized queueing. In our implementation, the Apache web server uses this mechanisms to implement the SRPT-based scheduling policy. Specifically, after determining the size of the requested file, Apache sets the priority of the corresponding socket by calling `setsockopt`. As Apache sends the file, the remaining size of the requested file decreases. When the remaining size falls below the threshold for the current priority class, Apache updates the socket priority with another call to `setsockopt`.

2.2.1 *Implementation Design Choices.*

Our implementation places the responsibility for prioritizing connections on the web server code. There are two potential problems with this approach. These are the overhead of the system calls to modify priorities, and the need to modify server code.

The issue of system call overhead is mitigated by the limited number of `setsockopt` calls which must be made. Typically only one call is made per connection. Even in the worst case, we make only as many `setsockopt` calls as there are priority classes (6 in our experiments) per connection.

A clean way to handle the changing of priorities totally within the kernel would be to enhance the `sendfile` system call to set priorities based on the remaining file size. We do not pursue this approach here as neither our version of Apache (1.3.14) nor Flash uses `sendfile`.

2.2.2 *Size cutoffs.* SRPT assumes infinite precision in ranking the remaining processing requirements of requests. In practice, we are limited to only 16 priority bands.

Based on experimentation, we have come up with some *rules-of-thumb* for partitioning the requests into priority classes which apply to the heavy-tailed web workloads. The reader not familiar with heavy-tailed workloads will benefit by first reading Section 5.

Denoting the cutoffs by $x_1 < x_2 < \ldots < x_n$:

—The lowest size cutoff x_1 should be such that about 50% of requests have size smaller than x_1. These requests comprise so little total load in a heavy-tailed workload that there's no point in separating them.

—The highest cutoff x_n needs to be low enough that the largest (approx.) .5% – 1% of the requests have size $> x_n$. This is necessary to prevent the largest requests from starving.

—The middle cutoffs are far less important. A logarithmic spacing works well.

In the experiments throughout this paper, we use only 6 priority classes to approximate SRPT. Using more improved performance only slightly.

2.2.3 *Priority to SYNACKs.* At this point one subtle problem remains: For requests for small files, a large portion of time to service the request is spent during the connection setup phase, *before* the size of the requested file is even known. The packets sent during the connection startup might therefore end up waiting in long queues, making connection startup very costly. For requests for small files, a long startup time is especially detrimental to response time. It is therefore important that the SYNACK be isolated from other traffic. Linux sends SYNACKs, to priority band 0. It is important when assigning priority bands to requests that we:

(1) Never assign any sockets to priority band 0.

(2) Make all priority band assignments to bands of *lower* priority than band 0, so that SYNACKs always have highest priority.

Observe that giving highest priority to the SYNACKs doesn't negatively impact the performance of requests since the SYNACKs themselves make up only a negligible fraction of the total uplink load.

Giving priority to SYNACKs is important in SRPT because without it the benefit that SRPT gives to small file requests is not noticeable. Later in the paper (Section 5.1) we consider whether the FAIR policy might also benefit by giving priority to SYNACKs, but find the improvement to FAIR to be less significant.

Assigning highest priority to SYNACKs has the negative effect of increasing the system's vulnerability to SYN-flooding attacks and severe overload. One possible solution to this problem is to take advantage of the fact that SYNACKs have their own priority band under SRPT and to monitor the rate of arrivals into this priority band. If the rate suddenly increases, indicating a potential SYN-flooding attack, we could drop the priority of SYNACKs, thus returning to a system closer to FAIR.

2.2.4 *The final algorithm.* Our SRPT-like algorithm is thus as follows:

(1) When a request arrives, it is given a socket with priority 0 (highest priority). This allows SYNACKs to travel quickly as explained in Section 2.2.3.
(2) After the size of the file requested is determined (by looking at the URL of the file), the priority of the corresponding socket is reset based on the size of the requested file, as shown in the table below.

Priority	Size (Kbytes)
0 (highest)	-
1	$\leq$ 1K
2	1K - 2K
3	2K - 5K
4	5K-20K
5	20K - 50K
6 (lowest)	$>$ 50K

(3) As the remaining size of the requested file diminishes, the priority of the socket is dynamically updated to reflect the remaining size.

3. LAN SETUP AND RESULTS

In Section 3.1 we describe the experimental setup and workload for the LAN experiments. Section 3.2 compares SRPT versus FAIR with respect to mean response time, in a LAN environment. Section 3.3 again compares SRPT versus FAIR in a LAN environment, but this time with respect to their performance on requests for large files. Finally Section 3.4 illustrates a simplification of the SRPT idea which involves only two priorities and yet still yields quite good performance.

3.1 LAN experimental setup

3.1.1 *Machine Configuration.* Our experimental setup involves six machines connected by a 10/100 Ethernet switch. Each machine has an Intel Pentium III 700 MHz processor and 256 MB RAM, and runs Linux 2.2.16. One of the machines is designated as the server and runs Apache 1.3.14. The other five machines act as web clients and generate requests as described below. Below we show results for both the case where the server uplink bandwidth is 10 Mbps and the case where the server uplink bandwidth is 100 Mbps. For the case of the 10 Mbps bandwidth, at any moment in time there may be a couple hundred simultaneous connections at the server. For the case of 100 Mbps bandwidth the number of simultaneous connections is in the thousands.

3.1.2 *Open versus closed systems.* To properly evaluate the performance of a server we need to understand how clients generate requests which drive the web server. The process by which clients generate requests is typically modeled either as an open system or as a closed system, as shown in Figure 3.

In an *open system* each user is assumed to visit the web site just once. The user requests a file from the web site, waits to receive the file, and then leaves. A request completion *does not* trigger a new request. A new request is only triggered by a new user arrival.

In a *closed system* model, it is assumed that there is some fixed number of users. These users sit at the same web site forever. Each user repeats these 2 steps, indefinitely: (i) request a file, (ii) receive the file. In a closed system, a new request is only triggered by the completion of a previous request.

When using a trace to generate requests under an open system model, the requests are generated at the times indicated by the trace, where interarrival times have been scaled to create the appropriate test load. When using a trace to generate requests under a closed system model, the arrival times of requests in the trace are ignored.

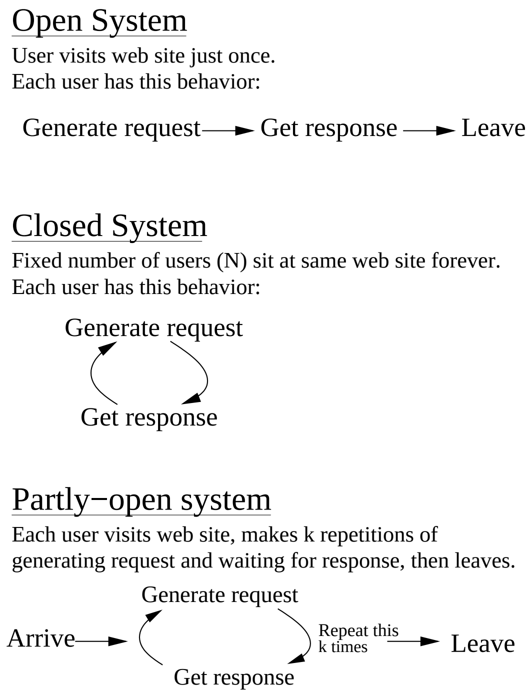


Fig. 3. *Three models for how the requests to a web server are generated. In all cases, every individual request averages into the mean response time.*

Neither the open system model nor the closed system model is entirely realistic. Throughout this paper we use the open system model. We also present results, however, for a different model which we call the *partly-open model*, which captures properties of both the open and closed models. Under the partly-open model, each user is assumed to visit a web site, make k requests for files at the web site, and then leave the web site. The k requests are made consecutively, with each request completion triggering the next request. We find that the results of the partly-open model are largely similar to those for an open model, see Figure 7.

In all the figures below, unless otherwise stated, we assume an open system model.

3.1.3 *Trace-based workload.* Throughout the paper we use a trace-based workload consisting of 1-day from the 1998 World Soccer Cup, obtained from the Internet Traffic Archive [ITA 2002]. The trace contains 4.5 million HTTP requests, virtually all of which are *static*. In our experiments, we use the trace to specify *the time* the client makes the request and the *size in bytes* of the file requested.

The entire 1 day trace contains requests for approximately 5000 different files. Given the mean file size of 5K, it is clear why all files fit within main memory and why the disk is not a bottleneck. Each experiment was run using a busy hour of the trace (10:00 a.m. to 11:00 a.m.). This hour consisted of about 1 million requests.

Some additional statistics about our trace workload: The minimum size file requested is a 41 byte file. The maximum size file requested is about 2 MB. The distribution of the file sizes requested fits a heavy-tailed truncated Pareto distribution (with α-parameter ≈ 1.2). The largest $< 3\%$ of the requested files make up $> 50\%$ of the total load, exhibiting a strong heavy-tailed property. 50% of files have size less than 1K bytes. 90% of files have size less than 9.3K bytes. The distribution of requested file sizes is shown in Figure 4.

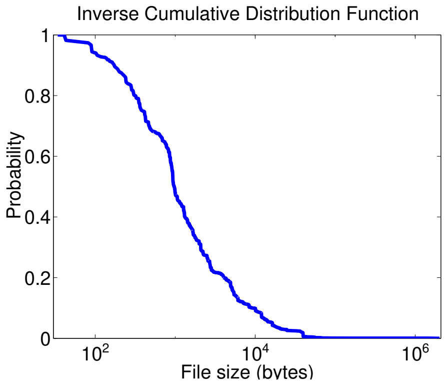
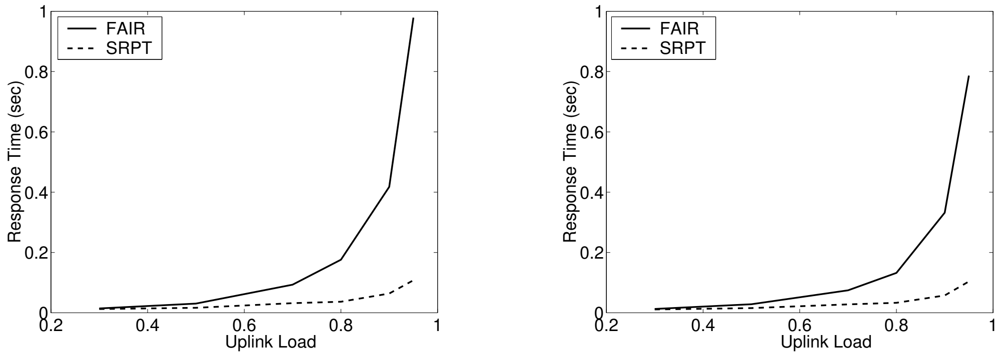


Fig. 4. *Inverse Cumulative Distribution Function, $\bar{F}(x)$, for the trace-based workload. $\bar{F}(x) =$ Pr$\{$size of the file requested $> x\}$*

We also repeated all experiments using a *web workload generator*, `Surge` [Barford and Crovella 1998] to generate the requests at the client machines. The `Surge` workload is created to be statistically representative of the file sizes at a web site, the sizes of files requested from a web site, the popularity of files requested, and more. We modified `Surge` simply to make it an open system. We have included in the associated technical report [HarcholBalter et al. 2000] the same set of results for the `Surge` workload. The `Surge` workload had a higher mean size of file requested (7K, rather than 5K), however in all other respects was statistically very similar to our trace-based workload. Not surprisingly, the factor improvement of SRPT over FAIR is very similar under the `Surge` and trace-based workloads. To be precise, all the response times for both FAIR and for SRPT are 50% higher under the `Surge` workload, and therefore the factor improvement is the same.



(a) Open system model

(b) Partly-open system model

Fig. 5. *Mean response time under SRPT versus FAIR as a function of uplink load, under trace-based workload, in LAN environment with uplink bandwidth 10 Mbps. (a) Assumes open system model (b) Assumes partly-open system model with $k = 5$ request-iteration cycles per user.*

3.1.4 *Generating requests at client machines.* In our experiments, we use `sclient` [Banga and Druschel 1999] for creating connections at the client machines. The original version of `sclient` makes requests for a certain file in periodic intervals. We modify `sclient` to read in traces and make the requests according to the arrival times and file names given in the trace. As in `sclient`, we assume a new connection for each request (no persistent connections).[2]

To create a particular uplink load, say 0.8, we simply scale the interarrival times in the trace's request sequence until the average number of bits requested per second is 8Mb/sec. We validate the uplink load both analytically and via measurement.

3.1.5 *Performance Metrics.* For each experiment, we evaluate the following performance metrics:

—*Mean response time.* The response time of a request is the time from when the client submits the request until the client receives the last byte of the requested file.

—*Mean slowdown.* The slowdown metric attempts to capture the idea that clients are willing to tolerate long response times for requests for large files and yet

[2]The effect of persistent connections on comparing FAIR and SRPT has been shown to be minimal. Under persistent connections, rather than using a new connection for each request, each connection opened by a client is reused for k requests, thus reducing the connection startup overhead of the TCP 3-way-handshake. Under a WAN setting with 100ms delay and 2.5% loss, both FAIR and SRPT benefit by a factor of 2 under persistent connections with $k = 5$. This effect is largely due to the savings in RTTs. Under a LAN setting, persistent connections are 50% more beneficial to FAIR as compared with SRPT because SRPT's 3-way-handshake time is already shorter, since SRPT gives priority to SYNACKs, and RTTs are not an issue.

expect short response times for small file requests. The slowdown of a request is therefore its response time divided by the time it would require if it were the sole request in the system. Slowdown is also commonly known as *normalized response time* or *stretch factor* and has been widely used [Bender et al. 1998; Roberts and Massoulie 1998; Downey 1997; Harchol-Balter and Downey 1997].

—*Mean response time as a function of the size of the file requested.* This metric indicates whether requests for large files are being treated *unfairly* under SRPT as compared with FAIR-share scheduling.

3.2 Mean improvements of SRPT under LAN

Before presenting the results of our experiments, we make some important comments.

—In all of our experiments the server's uplink bandwidth was the bottleneck resource. CPU utilization during our experiments remained below 5% for all the 10 Mbps experiments and below 80% for the 100 Mbps experiments, even for uplink load 0.95.

—The measured throughput and bandwidth utilization under the experiments with SRPT scheduling is *identical* to that under the same experiments with FAIR scheduling. The same exact set of requests complete under SRPT scheduling and under FAIR scheduling.

—There is no additional CPU overhead involved in SRPT scheduling as compared with FAIR scheduling. Recall that the overhead due to updating priorities of sockets is insignificant, given the small number of priority classes that we use.

Figure 5 shows the mean response time under SRPT scheduling as compared with the traditional FAIR scheduling as a function of uplink load. Figure 5(a) assumes that requests are generated according to an open model and Figure 5(b) assumes a partly-open system model, where each user generates $k = 5$ requests. Results are very similar in (a) and (b). For lower uplink loads the mean response times are similar under FAIR and SRPT. However for uplink loads > 0.5, the mean response time is a factor of $3 - 8$ lower under SRPT scheduling.

The performance results are even more dramatic for mean slowdown. Figure 6 shows the mean slowdown under SRPT scheduling as compared with the traditional FAIR scheduling as a function of load. For lower loads the slowdowns are the same under the two scheduling policies. For uplink load 0.5, the mean slowdown improves by a factor of 4 under SRPT over FAIR. Under an uplink load of 0.9, mean slowdown improves by a factor of 16.

Looking at the partly-open system model more closely we observe that mean response times are almost identical, regardless of the value of k. Figure 7 shows the performance of FAIR under a range of k values: $k = 1$, $k = 5$, and $k = 50$. It turns out that SRPT is even less sensitive to the choice of k.[3]

[3]Having experimented with many k values, we find the following subtle trend as we increase k: When we initially increase k, we find that response times drop a bit. The reason is that by synchronizing the times at which requests are generated, so that they are generated only when a previous request completes, we do a better job of evening the burstiness in the number of connections at the server. As k increases further, however, the partly-open system starts to look

Throughout we show results for the open system model, however we have verified that all these results are almost identical under the partly-open system model with $k = 5$.

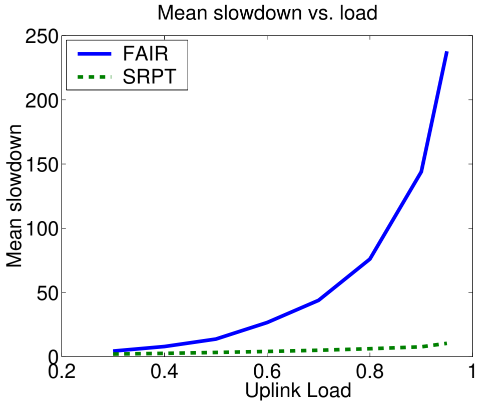


Fig. 6. *Mean slowdown under SRPT versus FAIR as a function of uplink load, under trace-based workload, in LAN environment with uplink bandwidth 10 Mbps. This slowdown plot corresponds to the experiment in Figure 5(a) and looks identical for the experiment shown in Figure 5(b).*

We conclude this section by once again considering the improvement of SRPT over FAIR, but this time in the case of a 100 Mbps uplink. Results are shown in Figure 8 under the Flash web server. We see that SRPT performs 5 times better than FAIR for an uplink load of 0.8, (i.e., 80 Mbps requested through a 100Mbps uplink). This is comparable to the factor improvement achieved in the case of the 10Mbps uplink under the Apache server, Figure 5(a).

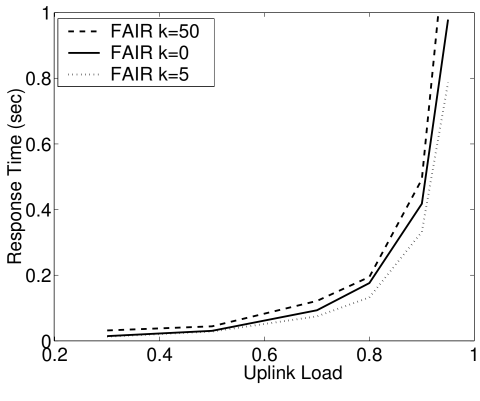


Fig. 7. *Performance of FAIR shown for a partly-open system model, using trace-based workload in a LAN with uplink bandwidth 10 Mbps, where $k = 1$, $k = 5$, and $k = 50$.*

---

like a closed system with zero think time. This has the effect of creating a near-one uplink load at all times, which causes response times to go up.

The significant improvements of SRPT over FAIR observed in this section are easily explained. The time-sharing behavior of FAIR causes small requests to be delayed in part by requests for large files, whereas SRPT allows requests for small files to jump ahead of requests for large files. Since most requests are for small files, most requests see an order of magnitude improvement under SRPT. Another way to think of this is that SRPT is an opportunistic algorithm which schedules requests so as to minimize the number of outstanding requests in the system (it always works on those requests with the least remaining work to be done). By minimizing the number of outstanding requests in the system, Little's Law [Little 1961] tells us that SRPT also minimizes the mean response time.

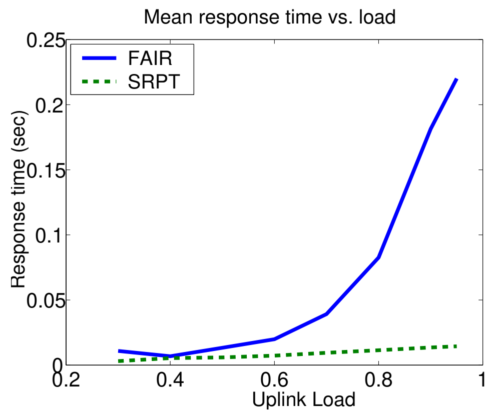
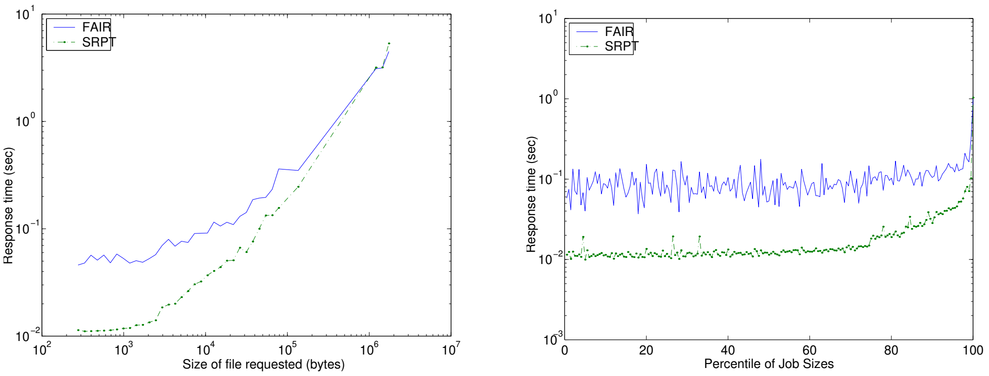


Fig. 8. *Mean response time under SRPT versus FAIR as a function of uplink load, under trace-based workload, in LAN environment with server uplink bandwidth 100Mb/sec.*

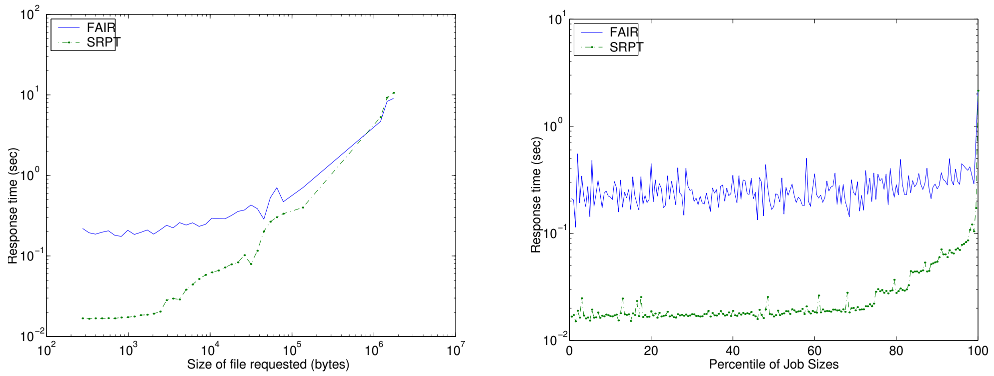
3.3 Performance of requests for large files under SRPT in LAN

The important question is whether the significant improvements in mean response time come at the price of significant unfairness to large requests. We answer this question for both the open system model and the partly-open system model. We first look at the case of 10 Mbps uplink and then at the case of 100 Mbps uplink.

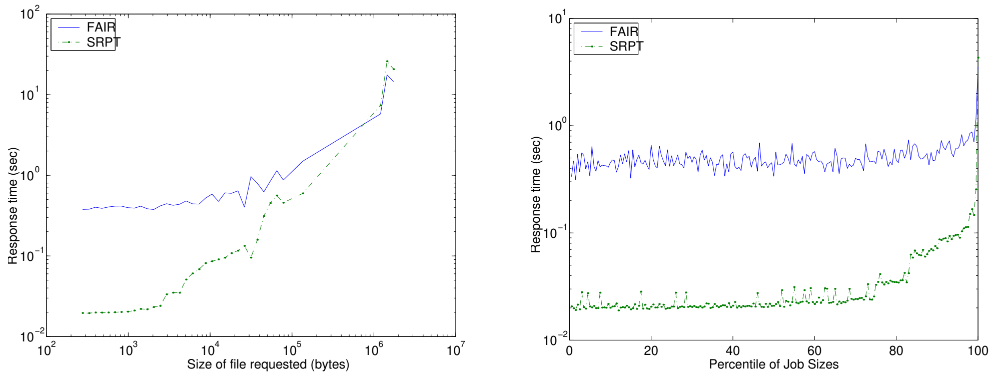
Figure 9 shows the mean response time as a function of the size of the requested file, in the case where the uplink load is 0.6, 0.8, and 0.9 and the bandwidth on the server's uplink is 10 Mbps. In the left column of Figure 9, the sizes of the files requested have been grouped into 60 bins, and the mean response time for each bin is shown in the graph. The 60 bins are determined so that each bin spans an interval $[x, 1.2x]$. Note that the last bin actually contains only requests for the very biggest file. Observe that requests for small files perform far better under SRPT scheduling as compared with FAIR scheduling, while requests for large files, those > 1 MB, perform only negligibly worse under SRPT as compared with FAIR scheduling. For example, under uplink load of 0.8 (see Figure 9(b)) SRPT scheduling improves the mean response times of requests for small files by a factor of close to 10, while the mean response time for the very largest size request only goes up by a factor of 1.2.



(a) uplink load = 0.6



(b) uplink load = 0.8



(c) uplink load = 0.9

Fig. 9. *Mean response time as a function of the size of the requested file under trace-based workload, shown for a range of uplink loads (corresponds to Figure 5(a)). The left column shows the mean response time as a function of the size of the file requested. The right column shows the mean response time as a function of the percentile of the requested file size distribution.*

Note that the above plots give equal emphasis to small and large files. As requests for small files are much more frequent, these plots are not a good measure of the improvement offered by SRPT. To fairly assess the improvement, the right column of Figure 9, presents the mean response time as a function of the percentile of the requested file size distribution, in increments of half of one percent (i.e. 200 percentile buckets). From this graph, it is clear that at least 99.5% of the requests benefit under SRPT scheduling. In fact, requests for the smallest 80% of files benefit by a factor of 10, and all requests outside of the top 1% benefit by a factor of > 5. For lower uplink loads, the difference in mean response time between SRPT and FAIR scheduling decreases, and the unfairness to requests for large files becomes practically nonexistent. For higher uplink loads, the difference in mean response time between SRPT and FAIR scheduling becomes greater, and the unfairness to requests for large files also increases. Even for the highest uplink load tested though (.95), there are only 500 requests (out of the 1 million requests) which complete later under SRPT as compared with FAIR. These requests are so large however, that the effect on their slowdown is negligible.

Results for the partly-open system model are similar to those in Figure 9, with slightly more penalty to the requests for large files, but still hardly noticeable penalty. For the case of $k = 5$, with uplink load $\rho = 0.8$, the mean response time for the largest 1% of requested files is still lower under SRPT (1.09 seconds under SRPT as compared with 1.12 seconds under FAIR). The request for the very largest file has a mean response time of 9.5 seconds under SRPT versus 8.0 seconds under FAIR.

For the 100 Mb/sec experiments *all requests*, large and small, preferred SRPT scheduling in expectation under all uplink loads tested.

3.4 SRPT with only two priorities

Our SRPT algorithm is only a rough approximation of true SRPT since we use only 6 priority classes. An interesting question is how much benefit one could get with only 2 priority classes. That is, each request would simply being of high priority or low priority.

To explore the performance of SRPT with only two priority classes, we define high-priority requests as those corresponding to the smallest 50% of files and low-priority requests as those corresponding to the largest 50% of files. The cutoff file size falls at 1K. We find that this simple algorithm results in a factor of 2.5 improvement in mean response time and a factor of 5 improvement in mean slowdown over FAIR. We also find that *all requests*, of either priority, have lower expected response times under SRPT than under FAIR using this simple algorithm.

4. WAN SETUP AND EXPERIMENTAL RESULTS

To understand the effect of network congestion, loss, and propagation delay in comparing SRPT and FAIR, we also conduct WAN experiments. We perform two types of WAN experiments: (i) experiments using our LAN setup together with a WAN emulator (Section 4.1) and (ii) experiments using physically geographically-dispersed client machines (Section 4.2). Throughout this section we use an uplink bandwidth of 10 Mbps.

4.1 WAN emulator experiments

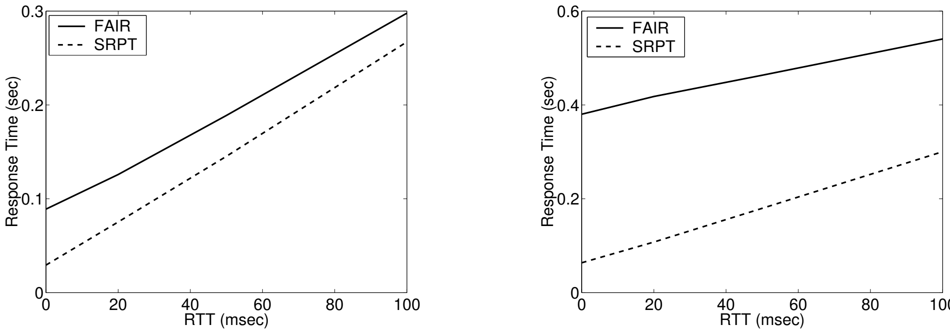
The two most frequently used tools for WAN emulation are probably NISTnet [NISTNet 2002] and Dummynet [Rizzo 1997].

NISTnet is a separate package available for Linux which can drop, delay or bandwidth-limit *incoming* packets. Dummynet applies delays and drops to both incoming and outgoing packets, hence allowing the user to create symmetric losses and delays. Since Dummynet is currently available for FreeBSD only we implement Dummynet functionality in form of a separate module for the Linux kernel. More precisely, we changed the `ip_rcv()` and the `ip_output()` function in the TCP-IP stack to intercept in- and out-going packets to create losses and delays.

In order to delay packets, we use the `timeout()` facility to schedule transmission of delayed packets. We recompile the kernel with `HZ=1000` to get a finer-grained millisecond timer resolution.

In order to drop packets we use an independent, uniform random loss model (as in Dummynet) which can be configured to a specified probability.

The experimental setup for our experiments is identical to that used for the LAN experiments (see Section 3.1) except that the WAN emulator functionality is now included in each client machine.

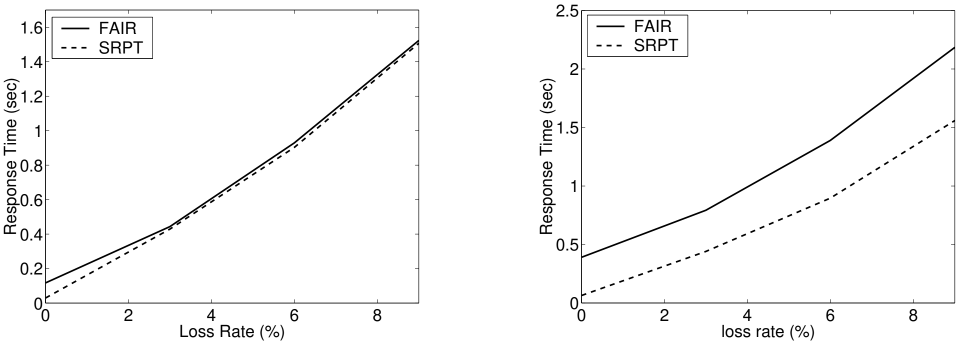


(a) uplink load = 0.7 (b) uplink load = 0.9.

Fig. 10. *Effect on SRPT and FAIR of increasing RTT from 0 ms to 100 ms.*

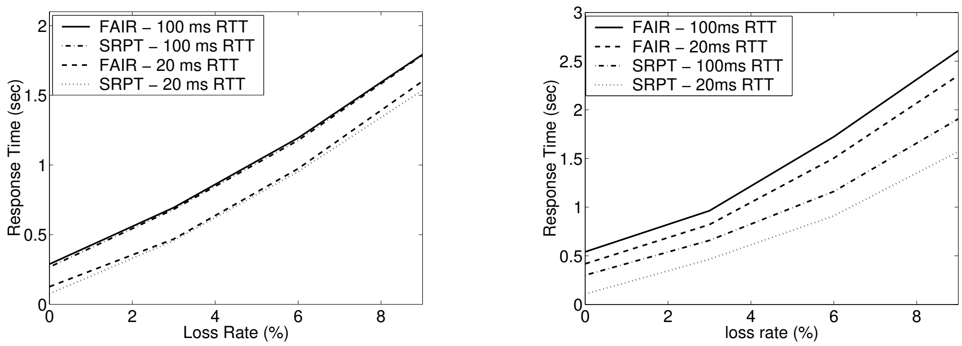
Figure 10 shows the effect of increasing the round-trip propagation delay (RTT) from 0 ms to 100 ms for FAIR and SRPT in the case of uplink load 0.7 and uplink load 0.9. Adding WAN delays increases response times by a constant additive factor on the order of a few RTTs for both FAIR and SRPT. The effect is that the relative improvement of SRPT over FAIR drops. Under uplink load $\rho = 0.9$, SRPT's improvement over FAIR drops from a factor of 4 when the RTT is 0 ms to a factor of 2 when the RTT is 100 ms. Under uplink load $\rho = 0.7$, the factor improvement of SRPT over FAIR drops from a factor of 2 to only 15%.

With respect to unfairness, we find that any unfairness to requests for large files decreases as the RTT is increased. The reason is obvious – any existing unfairness to requests for large files is mitigated by the additive increase in delay imposed on both FAIR and SRPT.



(a) uplink load = 0.7 (b) uplink load = 0.9.

Fig. 11. *Effect on SRPT and FAIR of increasing loss from 0% to 10%.*



(a) uplink load = 0.7 (b) uplink load = 0.9.

Fig. 12. *Effect on SRPT and FAIR of increasing loss and delay.*

Figure 11 assumes that the RTT is 0 ms and shows the effect of increasing the network loss from 0% to 10% under both FAIR and SRPT. Increasing loss has a more pronounced effect than increasing the RTT. We observe that the response times don't grow linearly in the loss rate. This is to be expected since TCPs throughput is inversely proportional to the square root of the loss. Under uplink load $\rho = 0.9$, SRPT's improvement over FAIR drops from a factor of 4 when loss is 0% to a factor of 25% when loss is 10%. Under uplink load $\rho = 0.7$, loss beyond 2% virtually eliminates any improvement of SRPT over FAIR.

With respect to unfairness, we find that loss slightly increases the unfairness to the request for the largest file under SRPT. The request for the largest file performs 1.1 times worse under 3% loss, but 1.5 times worse under loss rates up to 10%. Nevertheless, even in a highly lossy environment, the mean response time of requests for files in the top 1%-tile is still higher under FAIR as compared to SRPT.

Finally Figure 12 combines loss and delay. Since the effect of loss dwarfs the effect of propagation delay, the results are similar to those in Figure 11 with loss only.

4.2 Geographically-dispersed WAN experiments

We now repeat the WAN experiments using physically geographically-dispersed client machines. The experimental setup is again the same as that used for the LAN (see Section 3.1) except that this time the client machines are located at varying distances from the server. The table below shows the location of each client machine, indicated by its RTT from the server machine.[4]

Location	Avg. RTT
IBM, New York	20ms
Univ. Berkeley	55ms
UK	90-100ms
Univ. Virginia	25ms
Univ. Michigan	20ms
Boston Univ.	22ms

Unfortunately, we were only able to get accounts for Internet2 machines (schools and some research labs). The limitation in exploring only an Internet2 network is that loss and congestion may be unrealistically low.

Figure 13 shows the mean response time as a function of uplink load for each of the six hosts. The improvement in mean response time of SRPT over FAIR is a factor of 8–20 for high uplink load (0.9) and only about 1.1 for lower uplink load (0.5).

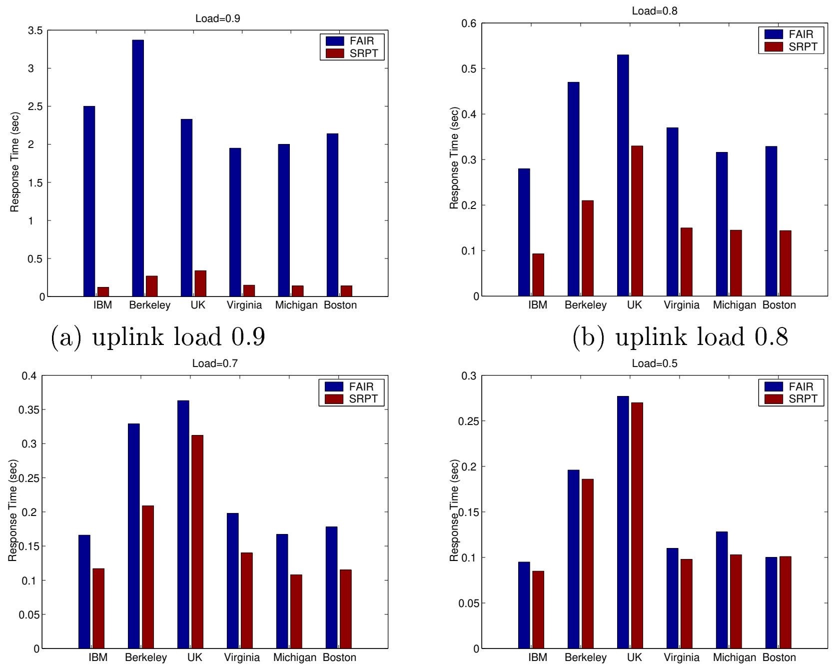
Figures 14(a) and 14(b) show the mean response time of a request as function of the percentile the size of the requested file, at an uplink load of 0.8, for the hosts at IBM and UK respectively. It turns out that *all* requests have higher mean response time under FAIR, as compared with SRPT. For the largest file, the mean response time is almost the same under SRPT and FAIR. The reason for the lack of unfairness is the same as that pointed out in the WAN emulation experiments for the case of significant RTT, but near-zero loss.

We next compare the numbers in Figure 13 with those obtained using the WAN emulation. For the case of uplink load 0.5, 0.7, and 0.8, the values of response time in Figure 13 are comparable with those obtained using the WAN emulator with propagation delay, but near-zero loss (compare with Figure 10).

Observe that the response times under uplink load 0.9 in Figure 13 are much higher than those for the WAN emulator for the case of FAIR but not for SRPT. The reason is that the WAN environment creates some variance in the uplink load. Thus an average uplink load of 0.9 translates to fluctuations ranging from 0.75 to 1.05, which means that there are moments of *transient overload*.[5] Transient overload affects FAIR far worse than SRPT because the buildup in number of requests at the server during overload is so much greater under FAIR than under SRPT. Transient overload even occasionally results in a full SYN queue under FAIR in our experiments. This means that incoming SYNs may be dropped, resulting in

[4]The measured bandwidth available at these client sites ranged from 1 Mbps, at Boston University, to 8 Mbps, at IBM. Experiments were instrumented such that the bandwidth at the client site would not be a bottleneck.

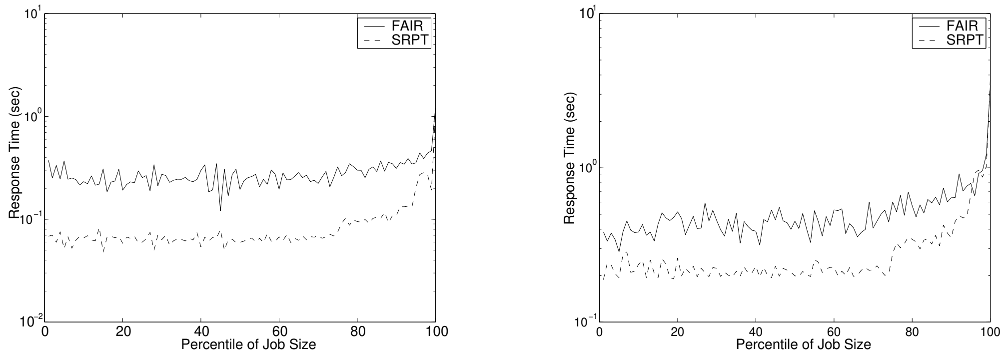
[5]When we say that there is a transient load of 1.05, we mean that during some 1-second intervals there may be 10.5 Mbits of data *requested* where the uplink bandwidth is only 10 Mbps.



(a) uplink load 0.9 (b) uplink load 0.8

(c) uplink load 0.7 (d) uplink load 0.5

Fig. 13. *Mean response time under SRPT versus FAIR in a WAN under uplink load (a) 0.9, (b) 0.8, (c) 0.7, and (d) 0.5.*



(a) IBM clients (b) UK clients

Fig. 14. *Response time as a percentile of the size of the requested file under SRPT scheduling versus traditional FAIR scheduling at uplink load 0.8, measured for (a) the IBM host and (b) the UK host.*

a timeout and retransmit. In the LAN environment where uplink load can be better controlled, we never experience SYN drops in our experiments (although SYN drops might occur in alternative setups where the CPU is the bottleneck resource).

The trends shown in Figures 13 and 14 are in agreement with the WAN emulator experiments. To summarize: (i) The improvement of SRPT over FAIR is higher at higher uplink loads; (ii) The improvement of SRPT over FAIR is diminished for far away clients; (iii) The unfairness to requests for large files under SRPT becomes non-existent as propagation delay is increased.

5. WHY DOES SRPT WORK?

In this section we look in more detail at where SRPT's performance gains come from and we explain why there is no starvation of requests for large files.

5.1 Where do mean gains come from?

The high-level argument has been given before: SRPT is an opportunistic algorithm which schedules requests so as to minimize the number of outstanding requests in the system (it always works on those requests with the least remaining work to be done). By minimizing the number of outstanding requests in the system, Little's Law tells us that SRPT also minimizes the mean response time: Little's Law [Little 1961] states that the mean number of requests in the system equals the product of the average arrival rate and the mean response time. In fact our measurements show that when the load is 0.7 the number of open connections is 3 times higher under FAIR than under SRPT. At load 0.9, this number jumps to 5 times higher. This corresponds to the improvement in mean response time of SRPT over FAIR.

Mathematically, the improvement of SRPT over FAIR scheduling with respect to mean response time has been derived for an M/G/1 queue in [Bansal and Harchol-Balter 2001].

At an implementation level, while our implementation of SRPT, described in Section 2.1 is not an exact implementation of the SRPT algorithm, it still has the desirable properties of the SRPT algorithm: requests for small files (or those with small remaining time) are separated from requests for large files and have priority over requests for large files. Note that our implementation does not interact illegally with the TCP protocol in any way: scheduling is only applied to those connections which are ready to send via TCP's congestion control algorithm.

The above discussion shows that one reason that SRPT improves over FAIR with respect to mean response times is because it allows small file requests to avoid time-sharing with large file requests. We now explore two other potential reasons for the improvement of SRPT over FAIR and eliminate both.

One potential reason for the improvement of SRPT over FAIR might be that FAIR causes the SYN queue to overflow (because of the rapid buildup in number of connections) while SRPT does not. Recall that if the web server's SYN queue fills up, new connection requests will experience expensive timeouts (on the order of 3 seconds). Our measurements show that the SYN queue is in fact significantly fuller under FAIR than under SRPT for high uplink loads, as expected, since SRPT minimizes the number of outstanding requests. However, in all of our experiments except one WAN experiment, the SYN queue *never* fills up under FAIR or SRPT.

Yet another potential reason for SRPT's performance gains over FAIR is that by having multiple priority queues SRPT is essentially getting to use more buffering, as compared with the single transmit queue of FAIR (see Figure 2). It is possible that there could be an advantage to having more buffering inside the kernel, since under high uplink loads we have observed some packet loss (5%) within the kernel at the transmit queue under FAIR, but not under SRPT. To see whether SRPT is obtaining an unfair advantage, we experimented with increasing the length limit for the transmit queue under FAIR from 100 to 500, and then to 700, entirely eliminating the losses. This helped just a little — reducing mean response time from about 400ms to 350ms under FAIR. Still, performance was nowhere near that of SRPT.

5.2 Why are requests for large files not hurt?

It has been suspected by many that SRPT is a very unfair scheduling policy for requests for large files. The above results have shown that this suspicion is false for web workloads. It is easy to see why SRPT should provide huge performance benefits for the requests for small files, which get priority over all other requests. In this section we describe briefly why the requests for large files also benefit under SRPT, *in the case of workloads with a heavy-tailed property.*

In general a heavy-tailed distribution is one for which

$$\Pr\{X > x\} \sim x^{-\alpha},$$

where $0 < \alpha < 2$. A set of file sizes following a heavy-tailed distribution has some distinctive properties:

(1) Infinite variance (and if $\alpha \leq 1$, infinite mean). (In practice, variance is not really infinite, but simply very high, since there is a finite maximum requested file size).

(2) The property that a tiny fraction (usually $< 1\%$) of the very longest requests comprise over half of the total uplink load. We refer to this important property as the **heavy-tailed property**.

The lower the parameter α, the more variable the distribution, and the more pronounced is the heavy-tailed property, *i.e.* the smaller the fraction of requests for large files that comprise half the uplink load.

The sizes of requested files have been shown to often follow a heavy-tailed distribution [Crovella and Bestavros 1997; Crovella et al. 1998]. Our traces have strong heavy-tailed properties. (In our trace the largest $< 3\%$ of the requests make up $> 50\%$ of the total uplink load.)

Consider a workload where the sizes of the files requested exhibit the *heavy-tailed property.* Now consider a request for a file in the 99%-tile of the requested file size distribution. This request will actually do much better under SRPT scheduling than under FAIR scheduling. The reason is that, under SRPT, this request only competes against 50% of the uplink load (the remaining 50% of the uplink load is made up of requests for the top 1%-tile of files) whereas it competes against 100% of the uplink load under FAIR scheduling. The same argument could be made for a requested file in the 99.5%-tile of the file size distribution.

However, it is not obvious what happens to a request in the 100%-tile of the requested file size distribution (i.e. the largest possible file). It turns out that, provided the uplink load is not too close to 1, the request in the 100%-tile will quickly see an idle period, during which it can run. As soon as the request gets a chance to run, it will quickly become a request in the 99.5%-tile, at which time it will clearly prefer SRPT. For a mathematical formalization of the above argument, in the case of an M/G/1 queue, we refer the reader to [Bansal and Harchol-Balter 2001].

Despite our understanding of the above theoretical result, we were nevertheless still surprised to find that results in practice matched those in theory – i.e., there was little if any unfairness to large requests. It is understandable that in practice there should be more unfairness to requests for large files since requests for large files pay some additional penalty for moving between priority queues.

6. PREVIOUS WORK

There has been much literature devoted to improving the response time of web requests. Some of this literature focuses on reducing network latency, e.g. by caching requests ([Gwertzman and Seltzer 1994], [Braun and Claffy 1994], [Bestavros et al. 1995]) or improving the HTTP protocol ([Padmanabhan and Mogul 1995]). Other literature works on reducing the delays at a server, e.g. by building more efficient HTTP servers ([Apache 2001], [Pai et al. 1999]) or improving the server's OS ([Druschel and Banga 1996], [Banga et al. 1998], [Kaashoek et al. 1996], [Mogul 1995]).

In the remainder of this section we discuss only work on *priority-based* or *size-based* scheduling of requests. We first discuss related implementation work and then discuss relevant theoretical results.

Almeida et al. [1998] use both a user-level approach and a kernel-level implementation to prioritizing HTTP requests at a web server. The *user-level* approach in [Almeida et al. 1998] involves modifying the Apache web server to include a scheduler process which determines the order in which requests are fed to the web server. This modification is all in the application level and therefore does not have any control over what the OS does when servicing the requests. The *kernel-level* approach in [Almeida et al. 1998] simply involves setting the priority of the process which handles a request in accordance with the priority of the request. Observe that setting the priority of a process only allows very coarse-grained control over the scheduling of the process, as pointed out in the paper. The user-level and kernel-level approaches in this paper are good starting points, but the results show that more fine-grained implementation work is needed. For example, in their experiments, the high-priority requests only benefit by 20% and the low priority requests suffer by up to 200%.

Another attempt at priority scheduling of HTTP requests which deals specifically with SRPT scheduling at web servers is that of Crovella et al. [1999]. This implementation does not involve any modification of the kernel. The authors experiment with connection scheduling at the *application level* only. They design a specialized Web server which allows them to control the order in which `read()` and `write()` calls are made, but does not allow any control over the low-level scheduling which occurs inside the kernel, below the application layer (*e.g.*, control over the order in which socket buffers are drained). Via the experimental Web server, the authors are able to improve mean response time by a factor of up to 4, for some ranges of uplink load, but the improvement comes at a price: *a drop in throughput by a factor of almost 2*. The explanation, which the authors offer repeatedly, is that scheduling at the application level does not provide fine enough control over the order in which packets enter the network. In order to obtain enough control over scheduling, the authors are forced to limit the throughput of requests. This is not be a problem in our paper. Since the scheduling is done at the kernel, we have absolute control over packets entering the network. Our performance improvements are greater than those in [Crovella et al. 1999] and do not come at the cost of any decrease in throughput.

The papers above offer coarser-grained implementations for priority scheduling of connections. Very recently, many operating system enhancements have appeared that allow for finer-grained implementations of priority scheduling [Radhakrishnan 1999; Almesberger 1999; Almesberger et al. 1999].

Several papers have considered the idea of SRPT scheduling in theory.

Bender et al. [1998] consider size-based scheduling in web servers. The authors reject the idea of using SRPT scheduling because they prove that SRPT will cause large files to have an arbitrarily high *max slowdown*. However, that paper assumes a worst-case adversarial arrival sequence of web requests. The paper goes on to propose other algorithms, including a theoretical algorithm which does well with respect to max slowdown and mean slowdown.

Roberts and Massoulie [1998] consider bandwidth sharing on a link. They suggest that SRPT scheduling may be beneficial in the case of heavy-tailed (Pareto) flow sizes.

Lastly, Bansal and Harchol-Balter [2001] investigate SRPT scheduling analytically for an M/G/1/SRPT queue (Poisson arrivals and general service times). We discussed these theoretical results in Section 5.

7. CONCLUSION AND FUTURE WORK

This paper demonstrates that the delay at a busy server can be greatly reduced by SRPT-based scheduling of the bandwidth that the server has purchased from its ISP. We show further that the reduction in server delay often results in a reduction in the client-perceived response time.

In a LAN setting, our SRPT-based scheduling algorithm reduces mean response time significantly over the standard FAIR scheduling algorithm. In a WAN setting the improvement is still significant for very high uplink loads, but is far less significant at moderate uplink loads.

Surprisingly, this improvement comes at no cost to requests for large files, which are hardly penalized, or not at all penalized. Furthermore these gains are achieved under no loss in byte throughput or request throughput.

Our current setup involves only *static* requests. In future work we plan to expand our technology to schedule cgi-scripts and other *non*-static requests. Determining the processing requirement of non-static requests is an important open problem, but companies are making excellent progress on predicting the run times of dynamic requests. We propose additionally to deduce the run time of a dynamic request as it runs. The request is initially assigned high priority, but its priority will decrease as it runs.

Our current setup considers the bottleneck resource at the server to be the server's limited bandwidth purchased from its ISP, and thus we do SRPT-based scheduling of that resource. In a different application (e.g. processing of cgi-scripts) where some other resource was the bottleneck (e.g., CPU), it might be desirable to implement SRPT-based scheduling of that resource.

Our current work also does not consider some practical aspects of implementing SRPT, such as vulnerability to denial of service attacks and other security issues.

Although we evaluate SRPT and FAIR across many uplink loads, we do not in this paper consider the case of overload. This is an extremely difficult problem both analytically and especially experimentally. Our preliminary results show that in the case of *transient overload* SRPT outperforms FAIR across a long list of metrics, including mean response time, throughput, server losses, etc.

Our SRPT solution can also be applied to server farms. Again the bottleneck resource would be the limited bandwidth that the web site has purchased from its ISP. SRPT-based scheduling could then be applied to the router at the joint uplink to the ISP or at the individual servers.

ACKNOWLEDGMENTS

We would like to thank Mark Crovella for many productive discussions on the topic of scheduling in web servers and Linux flow control. Greg Kesden, Jan Harkes, and Srini Seshan also provided us with helpful information about Linux internals. We would also like to thank Alexey Kuznetsov, Steven Ives and Tim Canfield for helping us with the initial Linux Diffserv experimentation. Lastly, we thank our many readers who gave us feedback on this paper including Ion Stoica, M. Satyanarayanan and Srini Seshan.

REFERENCES

ALMEIDA, J., DABU, M., MANIKUTTY, A., AND CAO, P. 1998. Providing differentiated quality-of-service in Web hosting services. In *Proceedings of the First Workshop on Internet Server Performance.*

ALMESBERGER, W. 1999. Linux network traffic control — implementation overview. White paper available at http://diffserv.sourceforge.net/.

ALMESBERGER, W., SALIM, J. H., AND KUZNETSOV, A. 1999. Differentiated services on Linux. White paper available at http://lrcwww.epfl.ch/linux-diffserv/.

APACHE. 2001. Apache web server. `http://httpd.apache.org`.

ARLITT, M., FRIEDRICH, R., AND JIN, T. 1999. Workload characterization of a web proxy in a cable modem environment. *ACM Performance Evaluation Review 27,* 2 (August), 25 – 36.

BANGA, G. AND DRUSCHEL, P. 1999. Measuring the capacity of a web server under realistic loads. *World Wide Web 2,* 1-2, 69–83.

BANGA, G., DRUSCHEL, P., AND MOGUL, J. 1998. Better operating system features for faster network servers. *ACM SIGMETRICS Performance Evaluation Review 26,* 3, 23–30.

BANSAL, N. AND HARCHOL-BALTER, M. 2001. Analysis of SRPT scheduling: Investigating unfairness. In *Proceedings of* ACM SIGMETRICS '01. 279 – 290.

BARFORD, P. AND CROVELLA, M. E. 1998. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of SIGMETRICS '98.* 151–160.

BENDER, M., CHAKRABARTI, S., AND MUTHUKRISHNAN, S. 1998. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms.*

Bestavros, A., Carter, R. L., Crovella, M. E., Cunha, C. R., Heddaya, A., and Mirdad, S. A. 1995. Application-level document caching in the Internet. In *Proceedings of the Second International Workshop on Services in Distributed and Networked Environments (SDNE'95).*

Braun, H. and Claffy, K. 1994. Web traffic characterization: an assessment of the impact of caching documents from NCSA's Web server. In *Proceedings of the Second International WWW Conference.*

Cockcroft, A. April 1996. Watching your web server. The Unix Insider at http://www.unixinsider.com.

Crovella, M. and Bestavros, A. 1997. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking 5,* 6 (December), 835–846.

Crovella, M., Frangioso, R., and Harchol-Balter, M. 1999. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems.*

Crovella, M., Taqqu, M. S., and Bestavros, A. 1998. Heavy-tailed probability distributions in the World Wide Web. In *A Practical Guide To Heavy Tails.* Chapman & Hall, New York, 3–26.

Downey, A. B. 1997. A parallel workload model and its implications for processor allocation. In *Proceedings of High Performance Distributed Computing.* 112–123.

Druschel, P. and Banga, G. 1996. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of OSDI '96.* 261–275.

Feldmann, A. Web performance characteristics. IETF plenary Nov.'99. http://www.research.att.com/~anja/feldmann/papers.html.

Gwertzman, J. and Seltzer, M. 1994. The case for geographical push-caching. In *Proceedings of HotOS '94.*

Harchol-Balter, M. and Downey, A. 1997. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems 15,* 3, 253–285.

HarcholBalter, M., Schroeder, B., Bansal, N., and Agrawal, M. 2000. Implementation of SRPT scheduling in web servers. Tech. Rep. CMU-CS-00-170.

ITA. 2002. The internet traffic archives. Available at `http://town.hall.org/Archives/pub/ITA/`.

Kaashoek, M., Engler, D., Wallach, D., and Ganger, G. 1996. Server operating systems. In *SIGOPS European Workshop '96.* 141–148.

Krishnamurthy, B. and Rexford, J. 2001. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement.* Addison-Wesley.

Little, J. 1961. A proof of the theorem $L = \lambda W$. *Operations Research 9*, 383 – 387.

Maggs, B. 2001. Personal communication with Vice President of Research, Akamai Technologies, Bruce Maggs.

Manley, S. and Seltzer, M. 1997. Web facts and fantasy. In *Proceedings of the 1997 USITS.*

Microsoft. 2001. The arts and science of Web server tuning with internet information services 5.0. Microsoft TechNet - Insights and Answers for IT Professionals: At http://www.microsoft.com/technet/.

Mogul, J. C. 1995. Network behavior of a busy Web server and its clients. Tech. Rep. 95/5, Digital Western Research Laboratory. October.

NISTNet. 2002. National institute of standards and technology. `http://snad.ncsl.nist.gov/itg/nistnet/`.

Padmanabhan, V. N. and Mogul, J. 1995. Improving HTTP latency. *Computer Networks and ISDN Systems 28*, 25–35.

Pai, V. S., Druschel, P., and Zwaenepoel, W. 1999. Flash: An efficient and portable Web server. In *Proceedings of USENIX 1999.*

Radhakrishnan, S. 1999. Linux – advanced networking overview version 1. Available at http://qos.ittc.ukans.edu/howto/.

Rizzo, L. 1997. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review 27,* 1.

Roberts, J. and Massoulie, L. 1998. Bandwidth sharing and admission control for elastic traffic. In *ITC Specialist Seminar.*

Schrage, L. E. and Miller, L. W. 1966. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research 14*, 670–684.

Silberschatz, A., Galvin, P., and Gagne, G. 2002. *Operating System Concepts, Sixth Edition.* John Wiley & Sons.

Stallings, W. 2001. *Operating Systems, Fourth Edition.* Prentice Hall.

Received February 2002; revised September 2002; accepted December 2002