Web servers under overload: How scheduling can help

Bianca Schroeder Mor Harchol-Balter *

Computer Science Department,

Carnegie Mellon University,

Pittsburgh, PA 15213

Abstract

This paper provides a detailed implementation study of the behavior of web servers serving static requests, where the load fluctuates over time (transient overload). Various external factors are considered including WAN delays and losses, and different client behavior models. We find that performance can be dramatically improved via a kernel-level modification to the web server to change the scheduling policy at the server from the standard FAIR (processor-sharing) scheduling to SRPT (Shortest-Remaining-Processing-Time) scheduling. We find that SRPT scheduling induces no penalties. In particular, throughput is not sacrificed and requests for long files experience only negligibly higher response times under SRPT than they did under the original FAIR scheduling.

1 Introduction

Most well-managed web servers perform well most of the time. Occasionally, however, every popular web server experiences transient *overload*. Overload is defined as the point when the demand on at least one of the web server's resources exceeds the capacity of that resource. While a well designed web server should not be persistently overloaded, transient periods of overload are often inevitable, since the traffic increase at the server that leads to the transient period of overload is difficult to predict. As an example, the amount of traffic received by a web site might rise because of an unexpected increase of the site's popularity, e.g. after being featured on national television or in

*This work was supported by NSF Career Grant CCR-0133077, by NSF ITR Grant 99-167 ANI-0081396, and by Spinnaker Networks via Pittsburgh Digital Greenhouse Grant 01-1.

a major newspaper. Another situation that could lead to transient periods of overload is under-provisioning for sales-boosting holidays. Although an online retailer knows to expect more web site hits during those days, it is difficult to predict exactly the associated increase in traffic volume.

An overloaded web server typically displays signs of its affliction within a few seconds. Work enters the web server at a greater rate than the web server can complete it. This causes the number of connections at the web server to build up. Very quickly, the web server reaches the limit on the number of connections that it can handle. From the client perspective, the client's request for a connection will either never be accepted or will get through only after several trials. Even when the client's request for a connection does get accepted, the time to service that request may be very long because the request has to *timeshare* with all the other requests at the server.

The solution most commonly suggested to avoid overload is admission control [18,24,34,61,62,64]: the web server or a front-end monitors the load and the capacity of the server and, if necessary, rejects incoming requests, in order to ensure satisfactory service to those requests being accepted to the server. The decision of when and which request to reject is based on sophisticated algorithms, e.g. leveraging application-level knowledge in the admission decision, applying techniques from control theory, or by combining admission control with QoS. Nevertheless, in the end it comes down to denying some customers service.

In this paper we suggest a different solution to server overload, that provides stable server performance, without dropping requests. Our solution is based on connection scheduling. Traditional web servers fairly proportion their resources among all requests (referred to as **FAIR** scheduling in the remainder of the paper). By contrast we propose *unfair* scheduling: instead of giving a fair share to each request, we give priority to requests for small files, in accordance with an approximation of the well-known scheduling algorithm preemptive Shortest-Remaining-Processing-Time-first (**SRPT**).

SRPT scheduling is not new. The SRPT algorithm makes sense in any context where the service requirement of jobs is known a priori. In such a context, the SRPT algorithm can be thought of as a greedy strategy to minimize the number of jobs in the system by always working on the job that is closest to completion. SRPT is provably the optimal scheduling policy for minimizing mean response times [57]. SRPT has been proposed for web systems serving static content before [31, 54], where it was observed that the service demand of a request is proportional to the size of the file requested (hence known a priori). However, SRPT has never been proposed in the context of overload, and has

typically only been evaluated under simulation [28, 47] or using application-level scheduling, which does not always provide sufficient scheduling control (see [21]).

Using SRPT under lower load situations is vindicated because of analytical work proving that SRPT does not hurt requests for long files, despite favoring requests for short files [10, 65].

What is new in this work is that we propose using SRPT as a solution for coping with transient overload. Overload is a regime where SRPT has never been evaluated (not analytically nor via simulation). There is a strong belief that long requests will starve under SRPT [13], [60] (p. 410), [59] (p. 162).

This paper makes two contributions:

First, we provide a detailed performance study of a web server under overload, showing just how bad overload can be. We experiment with both *persistent overload* (the request rate exceeds the server capacity during the entire experiment) and *transient overload* (alternate periods of overload and low load where the overall mean load is below 1). Load will be defined formally in the next section. We find that within a very short period, even low amounts of overload cause the server to experience instability, dropping requests from its SYN queue, while the client experiences very high response times (response time is defined as the time from when the client submits the request until receiving the last byte of the request). We evaluate a full range of complex environmental conditions, summarized in Table 2, including: the effect of WAN delay and loss, the effect of user aborts, the effect of persistent connections, the effect of SYN cookies, the effect of the RTO TCP timer, the effect of the packet length, and the effect of the SYN queue and ACK queue length limits.

Second, the paper proposes SRPT-like scheduling as a means to combat overload. We show that contrary to intuition, SRPT scheduling at the web server under transient overload does not unduly harm requests for large files as compared with traditional FAIR scheduling used in web servers. Furthermore SRPT scheduling significantly improves mean response times overall by up to an order of magnitude. Lastly, even the mean time until the first byte is improved by close to an order of magnitude under SRPT as compared with FAIR. Our implementation results are corroborated via theoretical approximations which match the trends we observe. Our evaluation of SRPT involves a *kernel-level implementation* whereby we *control the order in which the socket buffers at the server are drained into the network.*

The outline of this paper is as follows: In Section 2 we describe the experimental setup, including

the workload, the bottleneck resource, and how overload is generated. Section 3 explains our implementation of SRPT. Section 4 studies exactly what happens in a traditional (FAIR) web server under persistent overload and contrasts that with the performance of the modified (SRPT) web server. Section 5 compares the performance of the FAIR server and the SRPT server under transient overload. Section 6 analyzes where exactly SRPT's performance gains come from. Section 7 discusses relevant previous work, and Section 8 concludes.

2 Experimental Setup

2.1 Trace-based workload

In this paper we focus on servicing *static* requests, of the form "Get me a file." While web sites are increasingly generating dynamic content, studies from 1997-2001 [25, 37, 41] suggest that the request stream at web sites is still dominated by static requests. Even logs as recent as 2004 from proxy servers suggest that 67-73% of requests are for static content [32].

Serving static requests *quickly* is the focus of many companies *e.g.*, Akamai Technologies, and much ongoing research. In the context of overload, static requests are especially important, since popular web sites often convert their dynamic content into static content when the web site is overloaded [40].

The workload in our experiments is based on a 1-day trace from the Soccer World Cup 1998[1] obtained from the Internet Traffic Archive [30]. The trace contains 4.5 million mostly static HTTP requests. In our experiments, our clients generate requests according to an arrival process based on the interarrival times in the trace as explained in Section 2.3. The trace is also used to specify the request size in bytes. Results for experiments with other logs are given in the Appendix.

Some statistics about our trace workload follow: The mean file size requested is 5K bytes. The minimum size file requested is a 41 byte file. The maximum size file requested is a 2.02 MB file. The distribution of the file sizes requested has been analyzed in earlier work [4] and found to be *heavy-tailed*: while the body of the distribution can be reasonably well modeled by a log-normal

[1]The original trace contains 3 months of data. We choose a period of this trace that creates a load close to 1 in our experimental setup, to minimize the scaling necessary to achieve the different load levels we experiment with (see Section 2.3 on scaling). The period of the trace we choose exhibits the same statistical characteristics as the entire trace.

distribution, the tail is best fit by a Pareto distribution with an α-parameter less than 1.5. We find that the largest $< 3\%$ of the requests make up $> 50\%$ of the total load (in terms of total number of bytes), exhibiting a strong heavy-tailed property. 50% of files have size less than 1K bytes, and 90% of files have size less than 9.3K bytes. Figure 1 shows the full complementary cumulative distribution of requested file sizes.

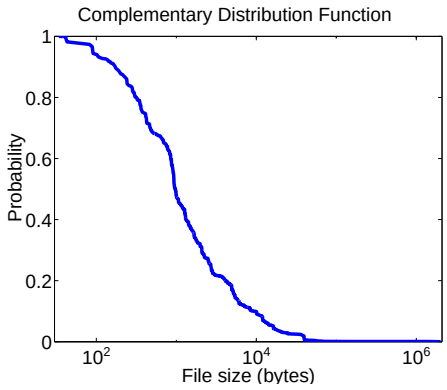


Figure 1: *Complementary Cumulative Distribution Function, $\bar{F}(x)$, for the trace-based workload.* $\bar{F}(x) = \Pr\{$*size of the file requested* $> x\}$

2.2 Load and the bottleneck resource

To understand the performance of a web server under overload, we need to understand which resource in a web server experiences overload 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 that the server has bought from its ISP [6, 19, 33]. Even a fairly modest server can completely saturate a T3 connection or 100Mbps Fast Ethernet connection. Also, buying more bandwidth is typically relatively more costly than upgrading memory or CPU. In fact, most web sites buy sufficient memory so that all their files fit within memory (keeping disk utilization low) [6]. For static workloads, CPU load is typically not an issue.

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 2. In all our experiments the bandwidth on the server's uplink is the bottleneck resource. **System load** is therefore defined in terms of the load on the server's uplink. 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 system load ρ 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.

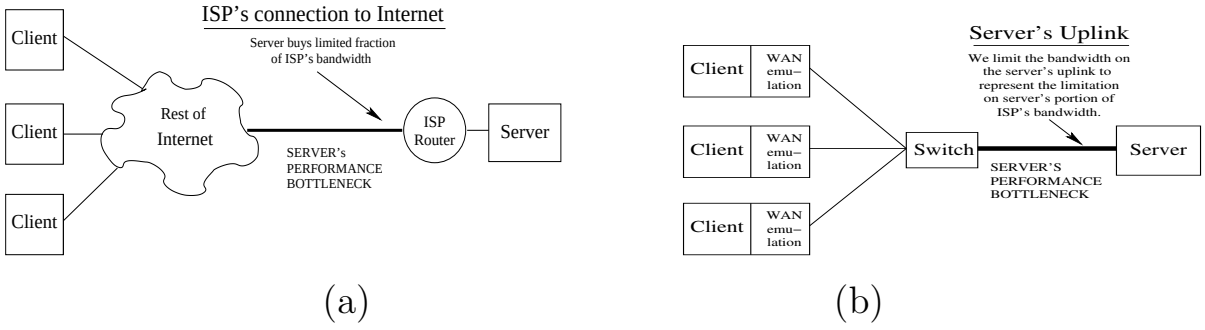


Figure 2: *(a) Server's bottleneck is the limited fraction of bandwidth that it has purchased from its ISP. (b) How our implementation setup models this bottleneck by limiting the server's uplink bandwidth.*

2.3 Defining persistent and transient overload

In our experiments we consider two types of overload, *persistent overload* and *transient overload*.

Persistent overload is used to describe a situation where the server is run under a fixed load $\rho > 1$ during the whole experiment. The motivation behind experiments with persistent overload is mainly to gain insight into what happens under overload. The overloaded state is unlikely to persist for too long in practice, due to system upgrades. Nevertheless, due to the burstiness of web traffic, even in the case of regular upgrades a popular web server is still likely to experience *transient* periods of overload.

We consider two different types of *transient overload*: In the first type, called *alternating overload*, the load alternates between overload and low load, where the length of the overload period is equal to the length of the low load period (see Figure 3(left)). In the second, called *intermittent overload*, the load is almost always low, but there are occasional "spikes" of overload, evenly spaced (see Figure 3(right)). *In all cases the overall mean system load is less than 1.*

Throughout, since the bandwidth on the uplink is the bottleneck resource, we define *load* to be the ratio of the bandwidth requested and the maximum bandwidth available on the uplink. To obtain a particular load we scale the interarrival times in the trace as follows: We first measure the system load (i.e. the bandwidth utilization) in an experiment using the original (unscaled) interarrival times

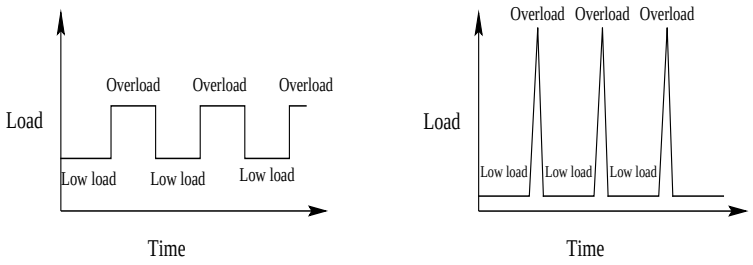


Figure 3: *Two different types of transient overload, alternating (left) and intermittent (right). Experimentally, the load will never look as constant as above, since arrival times and request sizes come from a trace.*

Workload	Type	Duration low load (seconds)	Duration overload (seconds)	Avg. low load	Avg. overload	Avg. load
W1	Alternating	25	25	$\rho = 0.2$	$\rho = 1.2$	0.7
W2	Alternating	10	10	$\rho = 0.2$	$\rho = 1.2$	0.7
W3	Alternating	50	50	$\rho = 0.2$	$\rho = 1.6$	0.9
W4	Alternating	25	25	$\rho = 0.4$	$\rho = 1.4$	0.9
W5	Alternating	10	10	$\rho = 0.4$	$\rho = 1.4$	0.9
W6	Alternating	40	40	$\rho = 0.1$	$\rho = 1.7$	0.9
W7	Intermittent	63	6	$\rho = 0.4$	$\rho = 5$	0.8
W8	Intermittent	63	10	$\rho = 0.45$	$\rho = 3$	0.8
W9	Intermittent	20	3	$\rho = 0.735$	$\rho = 2$	0.9
W10	Intermittent	13.3	2	$\rho = 0.735$	$\rho = 2$	0.9

Table 1: *Definition of trace-based workloads*

from the trace. In order to achieve a system load that is x times higher than the original load, we divide all interarrival times by x. Scaling the interarrival times (while keeping the sequence of requested web objects constant) models a general increase in traffic at the web server, e.g. due to sudden popularity or due to holiday shopping. Note that this type of overload is different from the case where a sudden rise in the popularity of one single object at a site causes overload. In this paper we are addressing only overload conditions that are caused by an increase in the arrival rate at the server, while the distribution of the documents requested remains the same (or similar) to that before overload.

We run all experiments in this paper for several different alternating and intermittent workloads, which are defined in Table 1. All our results are based on 30 minute experiments although we show only shorter fragments in the figures for better readability.

2.4 Why generating overload is difficult

Experimenting with persistent overload is inherently more difficult than running experiments where the load is high but remains below 1. The main issue in experimenting with overload is that running the server under overload is very taxing on *client* machines. While both the client machines and the server must allocate resources (such as TCP control blocks or file descriptors) for all accepted requests, the client machines must additionally allocate resources for all connections that the server has repeatedly refused (due to a full SYN-queue).

Requests to a web server may be generated either using an "open" system, a "closed" system, or some hybrid combination of the two, see e.g. [31]. To obtain overload, an open or partly-open system is necessary [7]. Existing web workload generators include `Surge` [11] and `Sclient` [7]. `Surge` mimics a closed system with several users, where each user makes a request and after receiving the response waits for a certain think time before it makes the next request. Note that in a closed system it is not possible to create overload, since a new request will be made only if another request finishes (see Figure 4(right)). By contrast, an open system allows one to create any amount of overload by simply generating a rate of requests where the sum of the sizes of the files requested exceeds the server uplink bandwidth.

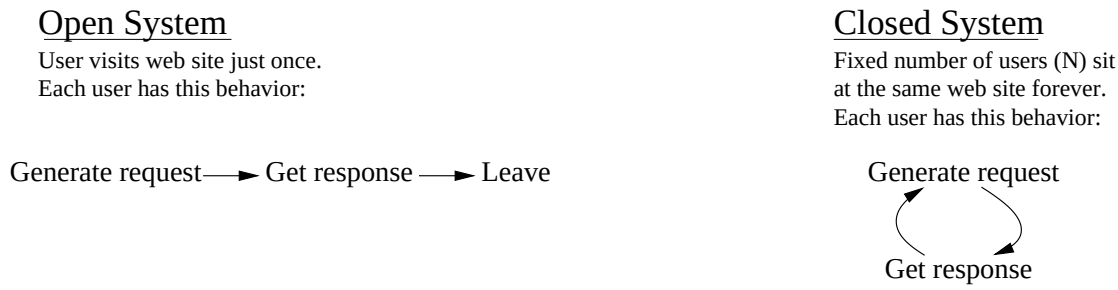


Figure 4: *Two models for how the requests to a web server are generated. In creating overload, one must use an open system model.*

Since none of the existing web workload generators support all the features we want to experiment with (open system, persistent connections, user abort and reload, etc.) we choose to implement our own trace-based web workload generator based on the `libwww` library [63]. `Libwww` is a client side web API based on *select()*. One obstacle in using `libwww` in building a web workload generator is that it does not support multiple parallel connections *to the same host*. We modify `libwww` in the following way to perform our experiments with persistent connections: Whenever our application

passes a URL to our modified `libwww`, it first checks whether there is a socket open to this destination that is (a) idle and (b) that has not reached the limit on the maximum number of times that we want to reuse a connection. If it doesn't find such a socket it establishes a new connection. We validate our workload generator by running experiments involving only the features supported by `Sclient`, and we find that `Sclient` and our new workload generator yield the same results. We also verify that the workload generator never becomes the bottleneck in the experiments, by checking that all requests are actually made at the desired times.

2.5 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 HTTP 1.1 requests as described in Section 2.1. The switch and the network cards of the six machines are forced into 10Mbps mode to make it easier to create overload at the bottleneck device.[2]

On the server machine we increased the size of the SYN and the ACK-queue to 512 as is common practice for high performance web servers We also increased the upper limit on the number of Apache processes from 150 to 350.[3]

Furthermore, we have instrumented the kernel of the server to provide detailed information on the internal state of the server. This includes the length of the SYN and ACK-queues, the number of packets dropped inside the kernel, and number of incoming SYNs that are dropped. We also log the number of active sockets at the server, which includes all TCP connections that have resources in the form of buffers allocated to them, except for those in the ACK-queue. Essentially, this means sockets being serviced by an Apache process, and sockets in the FIN-WAIT state.

Below we provide a brief tutorial of the processing of requests and sources of delays within a Linux-based web server.

A connection begins with a client sending a SYN. When the server receives the SYN it allocates

[2]Experiments were also performed under 100 Mbps mode, but not in overload since that puts too much strain on the client machines, see Section 2.4.

[3]Our FAIR server performed best with the above values: SYNQ = ACKQ = 512, and #Apache Processes = 350. The SRPT server is not affected at all by these limits, since SYNQ occupancy is always very low under SRPT.

an entry in the SYN-queue and sends back a SYN-ACK. After the client ACKs the server's SYN-ACK, the server moves the connection record to its ACK-queue, also known as the Listen queue. The connection waits in the ACK-queue until an Apache process becomes idle and processes it. Each Apache process can handle at most one request at a time. When an Apache process finishes handling a connection, the connection sits in the FIN-WAIT states until an ACK and FIN are received from the client.

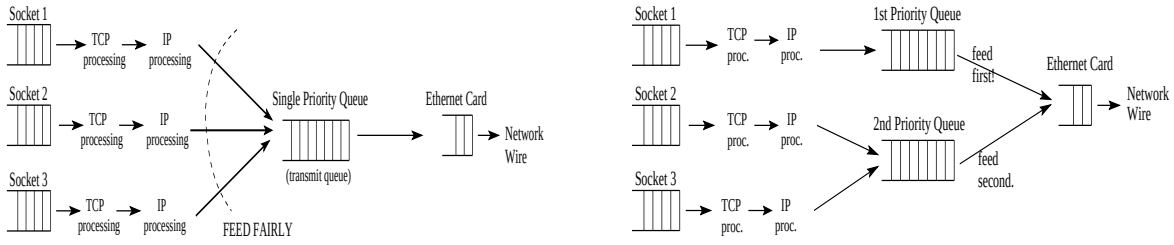
There are standard limits on the length of the SYN-queue (128), the length of the ACK-queue (128), and the number of Apache processes (150). These limits are often increased for high-performance web servers.

The limits above impose many sources of delays. If the server receives a SYN while the SYN-queue is full, it discards the SYN forcing the client to wait for a timeout and then retransmit the SYN. Similarly, if the server receives the ACK for a SYN-ACK while the ACK-queue is full, the ACK is dropped and must be retransmitted. The timeouts are long (typically 3 seconds) since at this time the client doesn't have an estimate for the retransmission timer (RTO) for its connection to the server. Lastly, if no Apache process is available to handle a connection, the connection must wait in the ACK-queue.

3 Implementing an SRPT web server

In this section we describe our implementation of SRPT in an Apache web server running on Linux.

3.1 Achieving priority queueing in Linux



(a) Standard Linux - FAIR

(b) Modified Linux - SRPT

Figure 5: *Data flow in standard Linux (left) and in Linux with priority queueing (right).*

Figure 5(a) shows the data flow in standard Linux. We will refer to the unmodified Apache server

running on this (unmodified) standard Linux as FAIR scheduling throughout.

There is a socket buffer corresponding to each connection. Data streaming into each socket buffer is then processed by TCP and IP. Throughout this processing, the packet stream corresponding to each connection is kept separate from every other connection. Finally, there is a *single*[4] "priority queue", into which *all* streams feed. Importantly, all streams get equal turns draining into the priority queue, subject to TCP windowing constraints. 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. We build the Linux kernel with support for the user/kernel Netlink Socket, QOS and Fair Queueing, and the Prio Pseudoscheduler. These kernel options allow us to switch the device queue from the default 3-band queue to a 16-band priority queue through the `tc` [2] user space tool.

Figure 5 (b) shows the flow of data in Linux after our kernel modifications. Again, the data is passed from the socket buffers through TCP and IP processing, during which the packet streams corresponding to each connection are kept separate. There are 16 priority queues (Figure 5 shows only 2). 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. The important point here is that *priority queue i is only allowed to drain if priority queues* 0 *through* $i-1$ *are all empty.* Note that since scheduling occurs after TCP/IP processing it does not interfere with TCP or IP.

3.2 Modifications to Apache

In order to approximate SRPT using the priority queues, for each request, we first have to initialize its socket to a priority corresponding to the requested file's size. The idea is to have sockets corresponding to smaller files drain into the higher priority queues. We later need to update the priority in agreement with the remaining size of the file. Both are done via the `setsockopt()` system call within the web server code.

The only remaining problem is that SRPT assumes infinite precision in ranking the remaining processing requirements of requests. In practice, we are limited to 16 priority bands.

It turns out that the way in which request sizes are partitioned among these priority levels is

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

somewhat important with respect to the server performance. We have experimentally derived some good *guidelines* that apply to the heavy-tailed web workloads. 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. Intuitively, the smallest 50% of requests comprise so little total load in a heavy-tailed distribution that there's no point in separating them further.

- The highest cutoff, x_n, needs to be low enough that the largest (approx.) 0.5% – 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 5 priority classes (cutoffs: 1KB, 2KB, 5KB, and 50KB) to approximate SRPT. Using more classes improved performance only slightly. While in our experiments we have hardcoded these size cutoffs in the Apache source code, they could be made accessible to the system administrator via a handle in the Apache configuration file. The system administrator or an automated mechanism could optimize the cutoffs if the file size distribution at a web site changes.

A potential problem with our approach is the overhead of the `setsockopt` system call used to modify priorities. However, this overhead is mitigated by the low system call overhead in Linux and the limited number of system calls: since there are only 5 priority classes the priority changes at most 4 times, and only for the very largest of the file requests.

4 Experimental Results – Persistent overload

In this section we study exactly what happens in a standard (FAIR) web server during persistent overload and contrast that with the performance of our modified (SRPT) server. We run the web server under persistent overload of 1.2, i.e., the average amount of data requested by the clients per second exceeds the bandwidth on the uplink by a factor of 1.2. We analyze our observations from two different angles, the server's view and the client's view.

We start with the server's view. One indication for the health of a server is the buildup in the number of connections at the server, shown in Figure 6(left). In FAIR, the number of connections

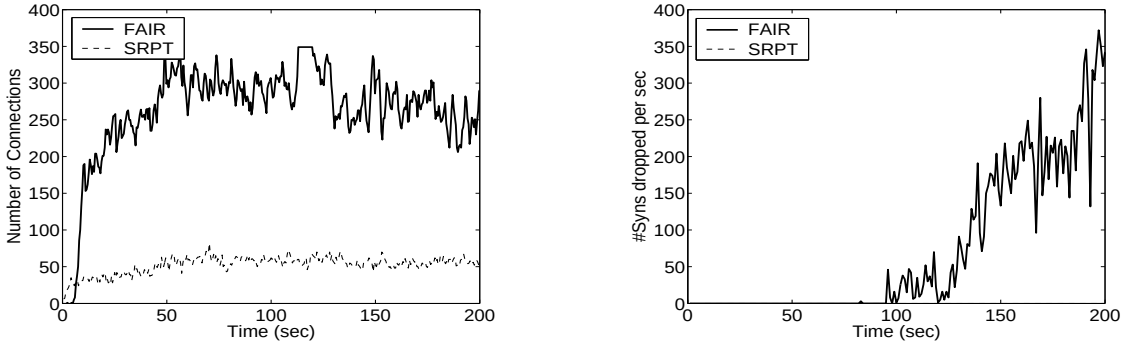


Figure 6: *Results for a persistent overload of 1.2 shown from the perspective of the server. (Left graph): Buildup in connections at the server; (Right graph): Number of incoming SYN packets dropped by the server.*

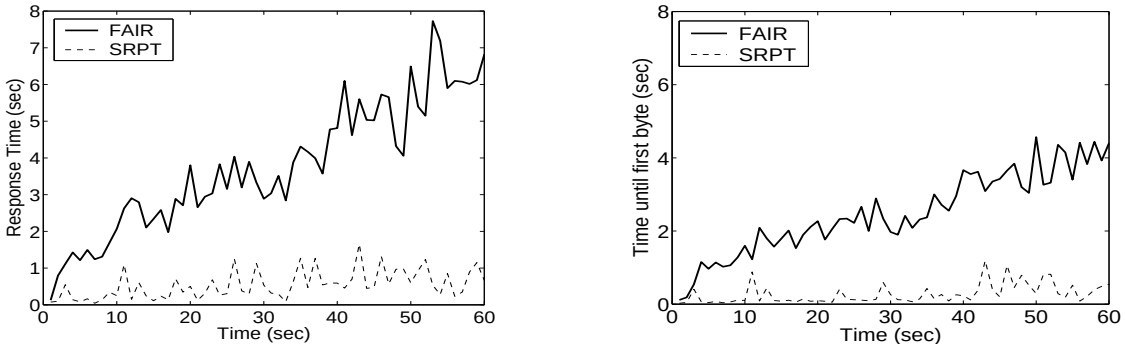


Figure 7: *Results for a persistent overload of 1.2 shown from the perspective of the client. (Left graph): Mean response time; (Right graph): Time until first byte is received.*

grows rapidly, until after around 50 seconds the number of connections reaches 350 – the maximum number of Apache processes. At this time all the Apache processes are busy, and consequently the SYN and the ACK queues fill up. After around 70 seconds the first incoming SYNs are dropped and the rate of SYN drops increases steadily and rapidly from that point on, as shown in (Figure 6(right)). By contrast, in the SRPT server the number of connections grows at a much slower rate. The reason is that the SRPT server queues up only requests for large files. Observe, that for an overload of 1.2 and an uplink capacity of 10 Mbps, each second the server is unable to complete 2 Mbit worth of requests on average. It turns out that largest requests compromising load between 1 and 1.2 have a mean size of about 1 Mbyte. Thus the SRPT server accumulates only one quarter of one request per second. After 200 seconds, it thus makes sense that the number of accumulated requests under SRPT is only 50, as shown in Figure 6(left). In fact our experiments show that the SRPT server does not start dropping requests until approximately the half hour mark.

Another server-oriented metric that we consider is *byte throughput*. We find that the byte throughput is the same under FAIR and SRPT . Despite the differences in the way FAIR and SRPT schedule requests, they both manage to keep the link fully utilized.

Next we describe the clients' experience in terms of *mean response time* and the *mean time until the first byte of a request is received*. In computing these metrics for persistent overload, we consider only those requests that finished before the final request arrived (subsequently, load will drop). Figure 7(left) shows that for the FAIR server response times grow rapidly over time. After only 40 seconds (long before the SYN-queue fills up) the mean response time is 5 sec, already intolerable. By contrast, under SRPT the response times are significantly lower and hardly grow over time. Figure 7(right) shows that the mean time until the first byte is received follows a trend very similar to that of the mean response time. We will therefore in the remainder of this paper use only the response time metric.

To understand the difference in response times between SRPT and FAIR under persistent overload we need to examine the effect of those requests that don't complete on the requests that do complete (and factor into the mean response time). Under FAIR, many of the requests that don't end up completing still steal a fair share of bandwidth from the other requests. Hence, they cause the response times of even requests for short files to increase. By contrast, under SRPT, all of the requests for the largest files (those that increase load from 1.0 to 1.2) do not receive any bandwidth under persistent overload: requests for small files are completely isolated from those for large files under SRPT. Thus, response times of the completing requests do not increase over time, even after tens of minutes.

To summarize the above observations, we see that after less than 100 seconds of very modest overload the FAIR server starts to drop incoming requests and the response times reach values that are not tolerable by users. The SRPT server significantly extends the time until SYNs are dropped and improves the client experience notably. We emphasize that the above experiments assumed very modest overload (as in the high load portion of workload W1 of Table 1). Some of the other workloads in Table 1 have more severe high loads. For example under a persistent high load of 1.4, we find that SYNs are dropped after only 18 seconds of persistent overload under FAIR, whereas SRPT avoids SYN drops for nearly half an hour of persistent overload. Response time growth is also much more dramatic under a persistent overload of 1.4.

5 Experimental Results – Transient overload

In this section we evaluate the performance of the standard (FAIR) web server and our modified (SRPT) server for *transient* overload. To level the playing field between SRPT and FAIR, we purposely choose to start by evaluating in detail the performance for the transient workload W1 from Table 1 (see Section 5.1). Workload W1 has the property that the duration and intensity of overload are modest (high load is only 1.2 for a duration of only 25 seconds), so that the SYN queue does not fill up under FAIR. Thus FAIR does not suffer the expensive timeouts under workload W1 due to a SYN drop, which appear under some of the other workloads. Our study considers all factors described in Table 2. In Section 5.2, we consider the other transient workloads in Table 1.

5.1 Results for workload W1

(A) The simple baseline case

In this section we study the simple baseline case described in Table 2, row (A). We first consider how the health of the server is affected during the low load and overload periods. We observe, as in the case of persistent overload, that the number of connections grows at a much faster rate under FAIR than under SRPT. While under SRPT the number of connections never exceeds 50, it frequently exceeds 200 under FAIR. However, neither server reaches the maximum SYN-queue capacity (since the overload period is short) and therefore no SYNs are dropped.

Figure 8(a) and (b) shows the response times over time of all jobs (averaged over 1 sec intervals) under FAIR and SRPT. These plots show just 200 sec out of a 30-minute experiment. The overload periods are shaded light gray while the low load periods are shaded dark gray. Observe that under FAIR the mean response times go up to more than 3 seconds during the overload period[5], while under SRPT they hardly ever exceed 0.5 sec.

Figure 8(c) shows the complementary cumulative distribution of the response times. Note that there is an order of magnitude separation between the curve for FAIR and SRPT. The mean response time taken over the entire length of the experiment is 1.1 sec under FAIR as compared to only 138

[5]Note that this is not quite as bad as for persistent overload, because a job arriving into the overload period in transient overload at worst has to wait until the low load period to receive service, so its expected response time is lower than under persistent overload.

Setup	Factor	Specific case shown in the left and middle columns of Figure 9 and Figure 10.	Range of values studied. Shown in rightmost column of Figure 9 and Figure 10.
(A)	Baseline Case	RTT=0, Loss=0%, No Persistent Conn., RTO=3 sec, packet length=1500, No SYN cookies, SYNQ=ACKQ=512, #ApacheProcesses=350	
(B)	WAN Delays	baseline + 100 ms RTT	RTT=0-150 ms
(C)	Loss	baseline + 5% loss	Loss=0-15%
(D)	WAN delay & loss	baseline + 100 ms RTT+ 5% loss	RTT=0-150 ms, Loss =0-15%
(E)	Persistent Connections	baseline + 5 req. per conn.	0-10 requests/conn.
(F)	Initial RTO value	baseline + RTO 0.5 sec	RTO = 0.5sec-3sec
(G)	SYN Cookies	baseline (SYN Cookies OFF)	SYN cookies=ON/OFF
(H)	User Abort/Reload	baseline + user aborts: User aborts after 10 sec and retries up to 3 times	Abort after 3-15 sec with up to 2, 4, 6, or 8 retries
(I)	Packet length	baseline + 536 bytes packet length	Packet length = 536-1500 bytes
(J)	Realistic Scenario	RTT=100 ms, Loss=5%, 5 req. per conn., RTO=3 sec, pkt. len.=1500, No SYN cookies, SYNQ=ACKQ=512, #ApacheProcs=350, User aborts after 7 sec and retries up to 3 times	

Table 2: *Columns 1 and 2 list the various factors. Column 3 specifies one value for each factor. This value corresponds to Figure 9(left, middle) and to Figure 10(left, middle). Column 4 provides a range of values for each factor. The range is evaluated in Figure 9(right) and in Figure 10(right).*

ms under SRPT. Furthermore, the variability in response times measured by the squared coefficient of variation is 6.39 under FAIR compared to 1.08 under SRPT. Another interesting observation is that the FAIR curve has bumps at regular intervals. These bumps are due to TCP's exponential backoff in the case of packet loss during connection setup. Given that we have a virtually loss-free LAN setup and the SYN-queue never fills up, one might wonder where these packets are dropped. Our measurements inside the kernel show that the timeouts are due to reply packets of the server being dropped inside the server's kernel. We will study the impact of this effect in greater detail in Section 6.

The big improvements in mean response time are not too surprising given the opportunistic nature of SRPT: schedule to minimize the number of connections. A more interesting question is what price large requests have to pay to achieve good mean performance.

This question is answered in Figure 8(d) which shows the mean response times as a function of the request size. We see that surprisingly *even the big requests hardly do worse under SRPT*. The very biggest request has a mean response time of 19.4 sec under SRPT compared to 17.8 sec under FAIR. If we look at the biggest 1 percent of all requests we find that their average response time is 2.8 sec under SRPT compared to 2.9 sec under FAIR, so these requests perform better on average under SRPT. We will explain this counter-intuitive result in Section 6.

(B) WAN delays

The two most frequently used tools for WAN emulation are probably NistNet [49] and Dummynet [56]. NistNet is a separate package available for Linux that 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 the form of a separate module for the Linux kernel. More precisely, we change the `ip_rcv()` and the `ip_output()` function in the Linux TCP-IP stack to intercept in- and out-going packets to create losses and delays.

In order to delay packets, we use the `add_timer()` facility to schedule the 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.

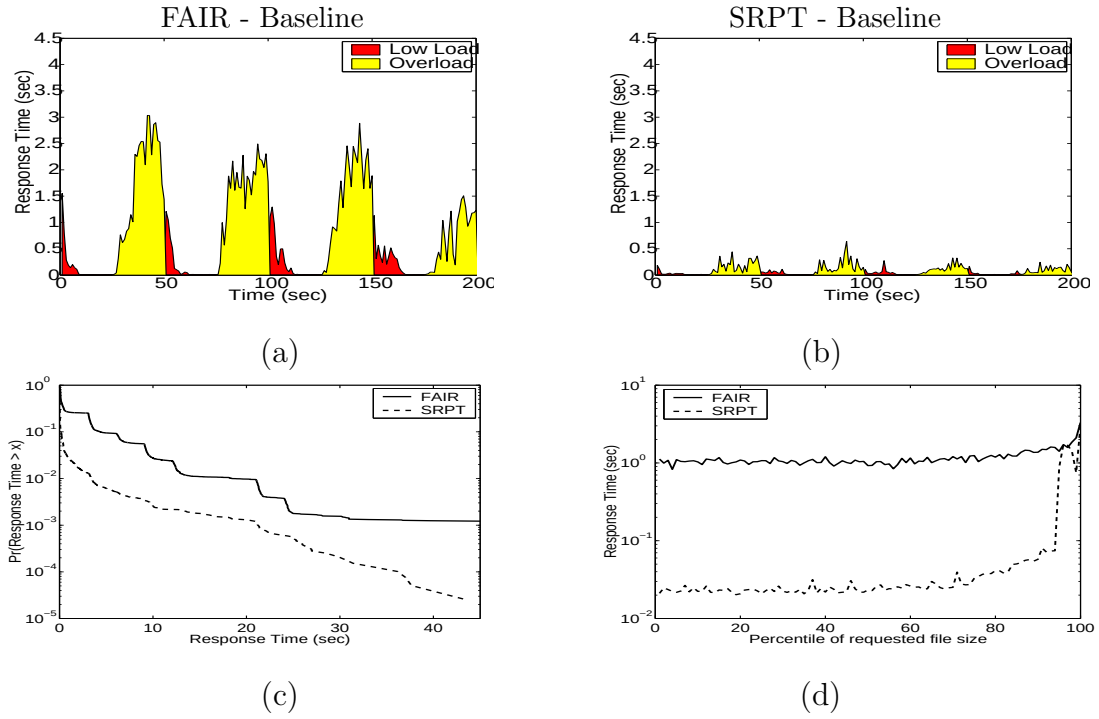


Figure 8: *Detailed performance under alternating workload W1 under the baseline case (setup (A)) of Table 2: (a) Mean response time under FAIR; (b) Mean response time under SRPT; (c) The complementary cumulative distribution of response times under FAIR and SRPT; (d) Response times as a function of the request size, showing requests for large files do not suffer.*

We experiment with delays between 0 and 150 msec and drop probabilities between 0 and 15%. This range of values was chosen to cover values used in related work [48] and values reported in actual live Internet measurements [55]. For example, for the month of October 2004 the Internet Traffic Report [55] reported maximum round-trip-times of 140 msec and maximum loss rates of 3.5%.

Figure 9(B)(left, middle) shows the effect of adding WAN delay (setup (B) in Table 2). This assumes a baseline setup, with an RTT (round-trip-time) delay of 100 msec. While FAIR's mean response time is hardly affected, since it is large compared to the additional delay, the mean response time of SRPT more than doubles.

Figure 9(B)(right) shows the mean response times for a range of RTTs from 0 to 150 msec. Observe that adding WAN delays increases response times by a constant additive factor on the order of a few RTTs. SRPT improves upon FAIR by at least a factor of 2.5 for all RTTs considered.

In the above experiments we assumed that all clients experience the *same* WAN delays. We also experimented with a heterogeneous environment, where each of the five client machines emulates

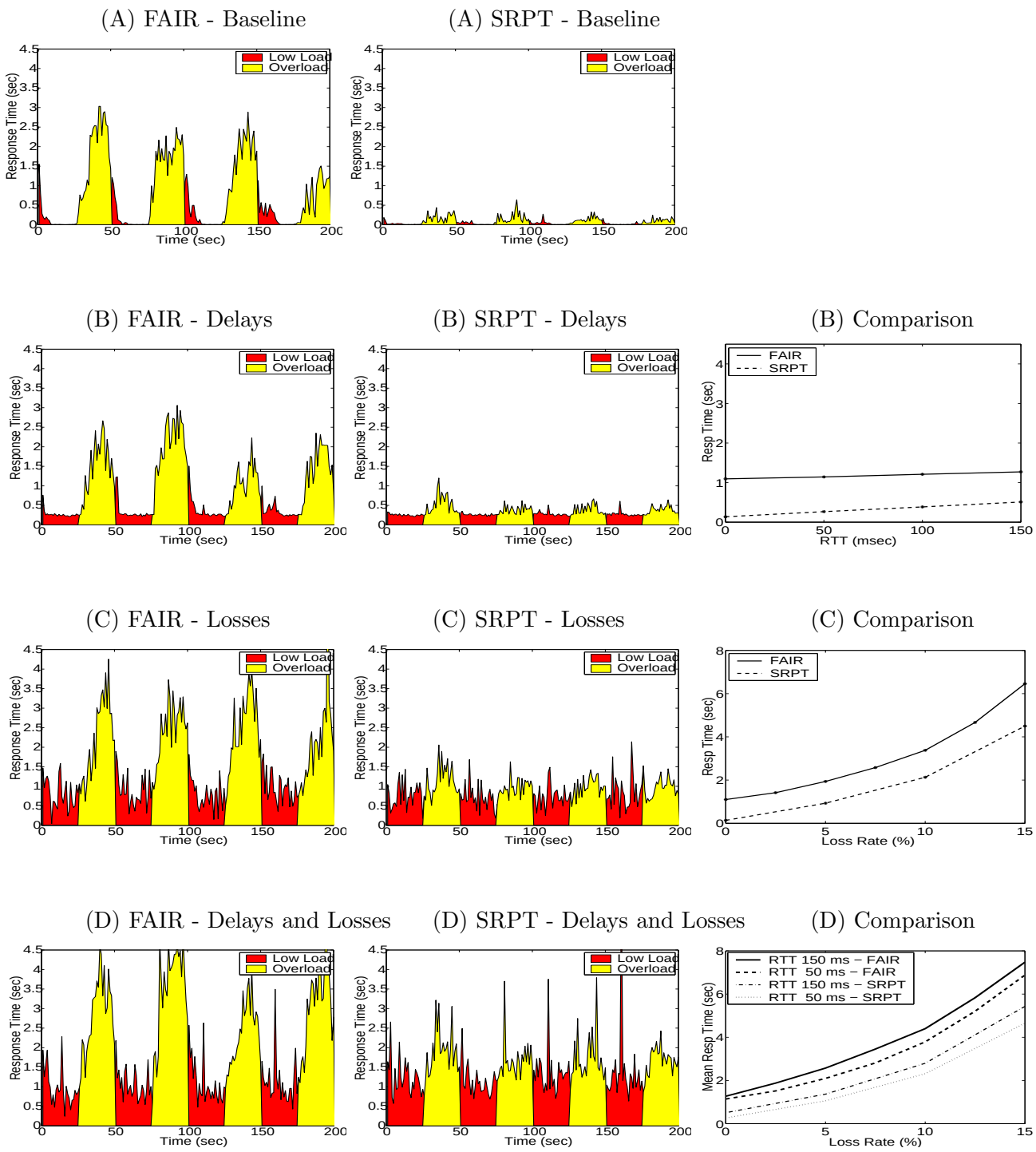


Figure 9: *Each row above compares SRPT and FAIR under workload W1 for one of the first four setups from Table 2. The left and middle columns show the response times over time for the specific values given in Table 2, column 3. The right column evaluates the range of values given in column 4 of Table 2.*

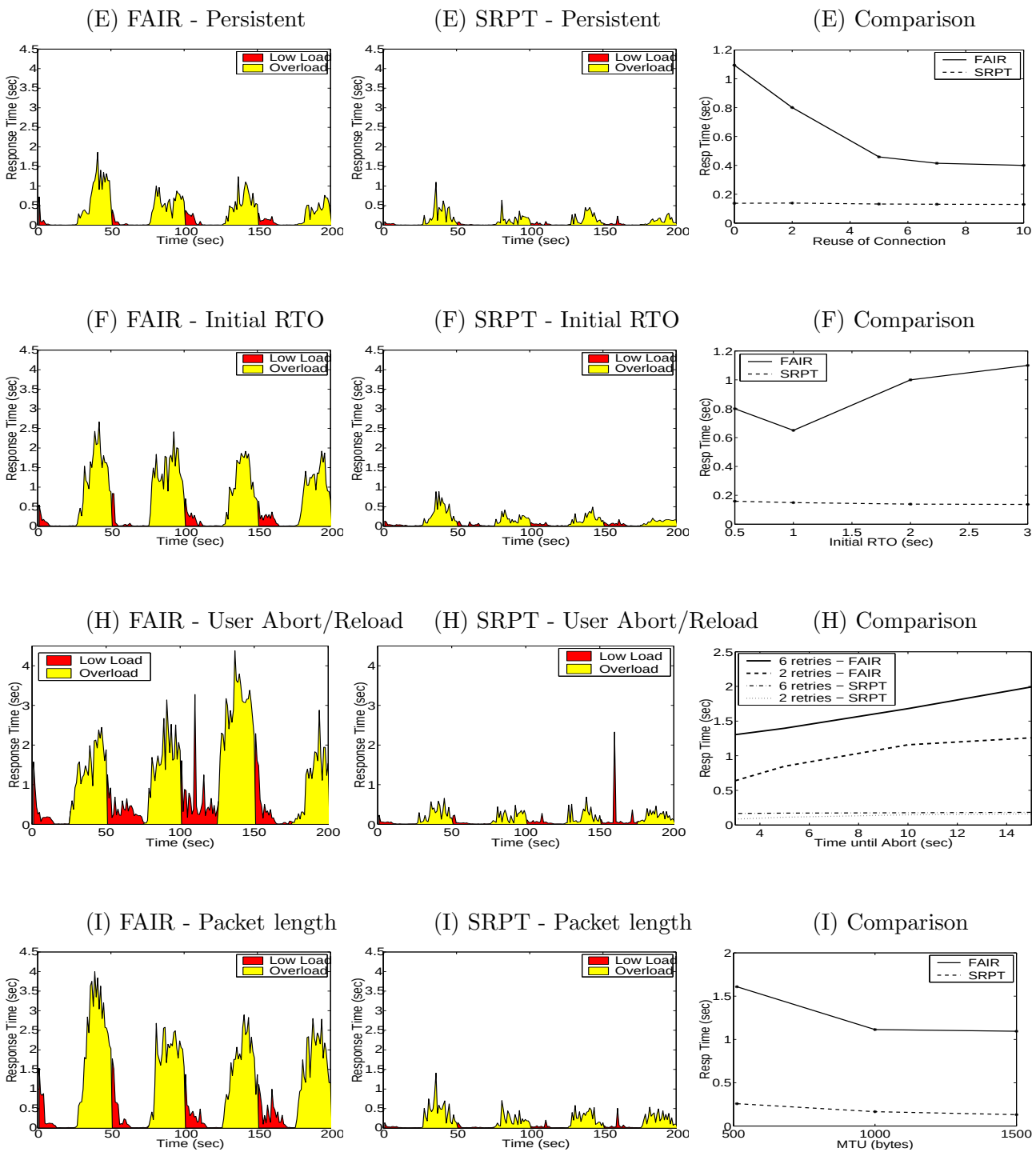


Figure 10: *Each row above compares SRPT and FAIR under workload W1 for one of setups (E), (F), (H), and (I). from Table 2. The left and middle columns show the response times over time for the specific values given in Table 2, column 3. The right column evaluates the range of values given in column 4 of Table 2.*

a different WAN delay ranging from 0 ms to 150 ms, where 150 ms represents our notion of the maximum end-to-end network delay. We find that the mean response time for a client with a given WAN delay equals that in an experiment where all clients emulate the same delay.

While one might think that in a heterogeneous environment, high delay clients might interfere with low delay clients, depriving them of the full benefits of SRPT scheduling, this is not the case for two reasons: First, under overload the time a connection stays alive at the server is mostly dominated by server delays, rather than by WAN delays; second, and most importantly, a slow connection will not "block" a fast connection: data for different connections is kept completely separate until after TCP/IP processing when it enters one of the priority queues shared by all connections (recall Figure 5). The data packets of a slow connection will therefore make it to the priority queues only after receiving the corresponding TCP ACKs (i.e. when the data is ready to be sent), and can therefore not slow down the fast connections (which will receive their ACKs at a fast rate). Thus the difference between SRPT and FAIR is minimized when all clients have WAN delay of 150 ms, and even here, the improvement factor of SRPT over FAIR is 2.5.

(C) Network losses

Figure 9(C)(left, middle) shows the mean response times over time for the setup in Table 2 row (C): a loss rate of 5%. In this case, for both FAIR and SRPT, the response times increase notably compared to the baseline case. FAIR's overall response time increases by almost a factor of 2 from 1.1 seconds to 1.9 seconds. SRPT's response time increases from less than 140 ms in the baseline case to around 930 ms.

Figure 9(C)(right) shows the mean response times for loss rates ranging from 0 to 15%. Note that the response times don't grow linearly with the loss rate. This is expected since TCPs throughput is inversely proportional to the square root of the loss [50].

Introducing frequent losses can increase FAIR's response times to more than 6 seconds and SRPT's response time to more than 4 seconds (as in the case of 15% loss). Even in this case SRPT still improves upon FAIR by about a factor of 1.5.

(D) Combination of Delays and Losses

Finally, we look at the combination of losses and delays. Figure 9(D)(left, middle) shows the results for the setup in Table 2 row (D): an RTT of 100 ms with a loss rate of 5%. Here SRPT improves upon FAIR by a factor of 2. Figure 9(D) (right) shows the response times for various combinations of loss rates and delays. We observe that the negative effect of a given RTT is accentuated under high loss rates. The reason is that higher RTTs make loss recovery more expensive since timeouts depend on the (estimated) RTT.

(E) Persistent Connections

Next we explore how the response times change if multiple requests are permitted to use a single *serial*, *persistent* connection ([44]) for several requests. Figure 10(E)(left, middle) shows the results for the setup in Table 2, row (E): where every connection is reused 5 times. Figure 10(E)(right) shows the response time as a function of the number of requests per connection, ranging from 0 to 10. We see that using persistent connections greatly improves the response times of FAIR. For example, reusing a connection five times reduces the response time by a factor of 2. SRPT on the other hand is hardly affected by using persistent connections. To see why FAIR benefits more than SRPT observe that reusing an existing connection avoids the connection setup overhead; this overhead is bigger under FAIR, mainly because it suffers from drops inside the kernel.[6] SRPT doesn't see this improvement since it experiences hardly any packet loss in the kernel.

Nevertheless, we observe that SRPT still improves upon FAIR by a factor of 3, even if up to 10 requests can use the same connection.

(F) Initial TCP RTO value

We observed previously that packets that are lost in the connection setup phase incur very long delays, which we attributed to the conservative initial RTO value of 3 seconds. We now ask how much of the total delay a client experiences in the FAIR server is due to the high initial RTO. To answer this question, we change the RTO value in Linux's TCP implementation. Figure 10(F)(left, middle) shows the results for the setup in Table 2, row (F): an RTO of 500ms. Figure 10(F)(right) explores the range from 500 ms up to the standard 3 seconds. Lowering the initial RTO can reduce FAIR's

[6]These drops occur when attempting to feed packets into an already full transmit queue (see Figure 5).

mean response time from originally 1.1 seconds (for the standard RTO of 3 seconds) to 0.65 seconds (for an initial RTO of 1 second). Reducing the initial RTO below 1 second introduces too much overhead due to spurious retransmissions, and therefore doesn't improve performance any further. SRPT's response times don't improve for lower initial RTOs since SRPT has little loss in the kernel. Nevertheless, for all initial RTO values considered, SRPT always improves upon FAIR by a factor of at least 4.

Having observed that the mean response times of a (standard) FAIR server can be significantly reduced by reducing TCP's initial RTO, we are not suggesting to change this value in current TCP implementations, since there are reasons for why it is set conservatively [52].

(G) SYN cookies

Recall that one of the problems under overload is the dropping of incoming packets due to a full SYN-queue. This leads us to the idea of using SYN cookies [14] in overload experiments. SYN cookies were originally developed to avoid denial of service attacks, however they have the added side-effect of eliminating the SYN-queue. (When using SYN cookies the server makes the SYN-ACK contents purely a function of the SYN contents. This way the SYN contents can be recomputed upon receipts of the next ACK, thereby avoiding the need for maintaining a SYN-queue.) Our hope is that by getting rid of the SYN-queue we will also eliminate the problems involving a full SYN-queue.

It turns out that the use of SYN cookies hardly affects the response times under workload W1 since in workload W1 the SYN-queue never fills up. In other transient workloads which exhibit SYN drops due to a full SYN-queue, the response times do improve, but only slightly by 2–5%. The reason is that now instead of the incoming SYN being dropped, it is the ACK for the SYN-ACK that is dropped due to a full ACK-queue.

Since SRPT does not lose incoming SYNs, its performance is not affected by using SYN cookies.

(H) User abort/reload

So far we have assumed that a user patiently waits until all the bytes for a request have been received. In practice, users abort requests after a while and hit the reload button of their browser. We model this behavior by repeatedly aborting a connection if it hasn't finished after a certain

number of seconds and then opening a new connection.

Figure 10(H)(left,middle) shows the response times in the case where a user waits for at most 10 seconds before hitting reload, and retrying this procedure up to 6 times before giving up. Figure 10(H)(right) shows mean response times if connections are aborted after 3 to 15 seconds and for either 2 or 6 retries.

We observe that taking user behavior into account can, depending on the parameters, sometimes increase and sometimes decrease response times. If users are impatient and abort connections quickly and retry only very few times, the response times can decrease by up to 10%. This is because there is now a (low) upper bound on the maximum response times and also the work at the server is reduced since clients might give up before the server has even invested any resources in the request. However, this reduced mean response time does not necessarily mean greater average user satisfaction, since some number of requests never complete. For example, if users abort a connection after 3 seconds and retry at most 2 times, more than 5 percent of the requests under FAIR never complete for workload W1.

If users are willing to wait for longer periods of time and retry more often, response times significantly increase. The reason is that response times are long both for those users that go through many retries, but also for other users, since frequent aborts and reloads increase the work at the server. For example, if users retry up to 6 times and abort a connection only after 15 seconds the response times under FAIR almost double compared to the baseline case that doesn't take user behavior into account. On the other hand the number of incomplete requests is very small in this case – under 0.02%.

For all choices of parameters for user behavior, the response times under SRPT improve upon those under FAIR by at least a factor of 8. Also the number of incomplete requests is always smaller under SRPT, by as much as a factor of 7. The reason is that SRPT is smarter than FAIR. Because SRPT favors small requests, the small requests (the majority) have no reason to abort. Only the few big requests are harmed under SRPT. Nevertheless, even requests for the longest file suffer no more incompletes under SRPT than under FAIR.

(I) Packet Length

Next we explore the effect of the maximum packet length (MSS). Two different packet lengths are

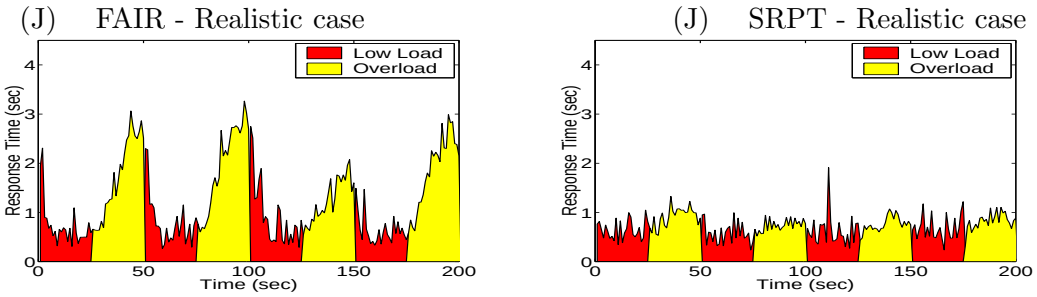


Figure 11: *Comparison of FAIR (left) and SRPT (right) for the realistic setup (J) for workload W1.*

commonly observed in the Internet [26]: 1500 bytes, since this is the MTU (maximum transmission unit) for Ethernet, and 536 bytes, which is used by some TCP implementations that don't perform MTU discovery [39].

Figure 10 (I)(left,middle) shows the results for setup (I) in Table 2, where the packet length is changed from the 1500 bytes in the baseline case to 536 bytes. As expected, the mean response time increases, since for a smaller packet length more RTTs are necessary to complete the same transfer. FAIR's response time increases by almost 50% to 1.6 seconds and SRPT's response time doubles to 260 msec.

Figure 10 (I)(right) shows the mean response times for different packet lengths ranging from 500 bytes to 1500 bytes. For all packet lengths considered, SRPT improves upon FAIR by at least a factor of 6.

(J) A realistic scenario

So far, we have looked at several factors affecting web performance in isolation. Figure 11 shows the results for an experiment that combines all the factors: We assume an RTT of 100 ms, a loss rate of 5%, no SYN-cookies, and the use of persistent connections (5 requests per connection). We leave the RTO at the standard 3 seconds and the MTU at 1500 bytes. We assume users abort after 7 seconds and retry up to 3 times.

We observe that the mean response time of both SRPT and FAIR increases notably compared to the baseline case. It is now 1.48 seconds under FAIR and 764 msec under SRPT – a factor 2 improvement of SRPT over FAIR. Observe that both these numbers are still better than those in experiment (D), where we combined only losses and delays and saw a response time of 2.5 seconds

and 1.2 seconds, respectively. The reason is that the use of persistent connections alleviates the negative effects of losses and delays during connection setup.

The largest 1% of requests (counting only those that complete) have a response time of 2.23 seconds under FAIR and 2.28 seconds under SRPT. Even the response time of the very biggest request is higher under FAIR: 13.2 seconds under FAIR and only 12.8 seconds under SRPT. The total number of incomplete requests (those aborted the full three times) is the same under FAIR and SRPT – about 0.2%.

Observe that it makes sense that unfairness to large requests is more pronounced in the baseline case because server delay dominates response time under the baseline case, in contrast to the realistic case where external factors can dominate.

Under workload W1 there were no SYN drops, neither for the baseline nor the realistic setup.

5.2 Other transient workloads

Figure 12 gives the mean response times for all ten workloads from Table 1 for the baseline case (Table 2 row (A)). While Figure 12 depicts *mean* response times over the entire duration of the experiment, it is important to notice that *peak* response times are actually much higher. For example, for workload W1, when considering the response times *averaged over 1 second intervals* as in Figure 8(a) and (b), we witnessed response times three times higher than that shown in Figure 12 for workload W1. This underscores the need for SRPT scheduling.

The reason that we choose to show performance for the baseline case rather than the realistic case is that this way the effects of the different workloads are not blurred by external factors. Also, the starvation of large requests is by definition greater for the baseline case than the realistic case, as explained above. In the discussion below, however, we will include the performance numbers for both the baseline and the realistic case.

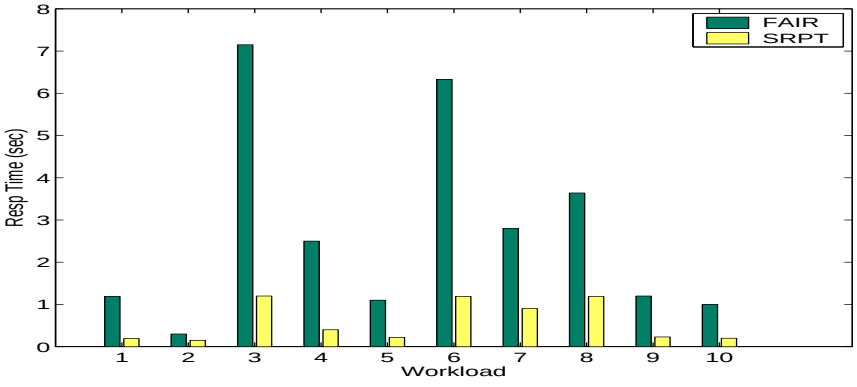
Mean response time

For the *baseline case*, the mean response time of SRPT improves upon that of FAIR by a factor of 2 – 8 across the ten different workloads. More specifically, FAIR ranges from 300ms – 7.2 seconds, while SRPT ranges from 150ms – 1.2 seconds.

Under the *realistic case* (not shown), SRPT improves over FAIR by a factor of 1.4 – 4 across

workloads W1, W2, W4, W5, W7, W8, W9, and W10. Note that we exclude workloads W3 and W6 throughout the discussion of the realistic scenario. This is because these two workloads have such a long overload period, that under the realistic scenario, their behavior mimics persistent overload. To see this, observe that, under the realistic scenario, which involves user aborts, there is an increased load due to multiple aborts and retries, which are especially prevalent in W3 and W6 because of their long overload period. This increased load causes the time-average load in these workloads to rise from 0.9 to over 1.0. Hence the system is running in persistent overload.

For both the baseline and realistic case, the mean performance under each workload is affected by (1) the mean system load and (2) the length of the overload and low load periods. Point (2) should be clear from looking at Figure 12, where workloads W3 and W6, which have the longest periods of overload, both stand out. Point (2) is also justified by considering workloads W1 and W2 which only differ in the length of their overload periods, resulting in a factor 5 difference in their mean response time, or workloads W4 and W5 which also differ only in the length of their overload period, again resulting in a factor 2 difference in their mean response times. The length of the overload period and the overall load also affect the number of SYN drops, and consequently the response times. While about half the workloads have zero SYN drops under FAIR and SRPT, under both the baseline and realistic setups, workload W3 which has 50 seconds of overload, results in 50% SYN drops under FAIR (zero SYN drops under SRPT), and consequently very high mean response times. SYN drops do not occur under SRPT under any of our workloads.



Mean Response Time - Baseline

Figure 12: *Comparison of mean response times under FAIR and SRPT in the baseline case for the workloads in Table 1. Peak response times are far worse than mean response times.*

Performance of large requests

Again, an obvious question to ask is whether this improvement in the mean response time comes at the price of longer response times for the requests for large files. In particular, for the workloads with higher mean system load and longer overload periods it might seem likely that SRPT leads to a higher penalty for the long requests.

We find that these concerns are unfounded. The mean response times of only the biggest 1% of all requests is never more than 10% higher under SRPT than FAIR for any workload in the *baseline case*, and is often lower under SRPT, and this penalty is further substantially diminished under the *realistic scenario*.

When considering the performance of large requests in the case of the *realistic setup*, it is important to also look at the number of incomplete requests (incomplete requests are only a consequence of user aborts). We observe that the lack of unfairness under SRPT in the realistic setup is not a consequence of a large number of incomplete large requests. The overall fraction of incomplete requests is only 0.2% for both FAIR and SRPT when load is 0.7 and ranges from 10 – 15% for both FAIR and SRPT when load is 0.9 (again excluding workloads W3 and W6). Looking only at the largest 1% of requests, the fraction of incomplete requests is much more variable, ranging from 3 – 25% under SRPT and 3 – 30% under FAIR, but is typically smaller under SRPT than under FAIR.

Finally, we observe that for both the baseline and the realistic setup, increasing the length of the overload period or the mean system load does *not* result in more starvation. This also agrees with the theoretical M/GI/1 results in [9].

6 Why does SRPT work?

In this section we will look in more detail at where SRPT's performance gains come from and we explain why there is no starvation of long jobs.

6.1 Where do mean gains come from?

Recall that we identified in Section 4 and Section 5 three reasons for the poor performance of a standard (FAIR) server under overload:

1) High queueing delays at the server due to high number of connections sharing the bandwidth,

see Figure 6 and Figure 7.

2) Drops of SYNs because of full SYN queue,

3) Loss of packets inside the kernel.

SRPT alleviates all these problems. It reduces queueing delays by scheduling connections more efficiently. It has a lower number of dropped SYNs since it takes longer for the SYN-queue to fill. Finally and less obviously, an SRPT server also sees less loss inside the kernel. The reason is that subdividing the transmit queue into separate shorter queues allows SRPT to isolate the short requests, which are most requests. These short requests go to a queue which is drained more quickly, and thus experience no loss in their transmit queue.

The question we address in this section is how much of SRPT's performance gain can be attributed to solving each of the above three problems.

We begin by looking at how much of the performance improvements under SRPT stem from alleviating the SYN drop problem. In workload W1, used throughout the paper, no incoming SYNs were dropped. Hence SRPT's improvement was not due to alleviating SYN drops. Workload W4 did exhibit SYN drops (1% in the baseline case and 20% in the realistic case). We eliminate the SYN drop advantage by increasing the length of the SYN-queue to the point where SYN drops are eliminated. This only improves FAIR by under 5% for the baseline case and 30% for the realistic case – not enough to alone account for SRPT's big improvement over FAIR.

The remaining question is how much of SRPT's benefits are due to reducing problem 1 (queueing delays) versus problem 3 (packet drops inside the kernel). Observe that problem 3 is mainly a problem because of the high initial RTO. We can mitigate the effect of problem 3 by dropping the initial RTO to, say, 500 ms. The result is shown in Figure 10(F). Observe that even when problem 3 is removed, the improvement of SRPT over FAIR is still a factor of 7[7].

We therefore conclude that problem 1 is the main reason why SRPT improves upon FAIR. By timesharing among many requests, the FAIR scheduling policy ends up slowing down all the requests.

[7]Problem 3 may seem to be a design flaw in Linux, that could be solved by either adding a feedback mechanism when writing to the transmit queue or by increasing the length of the transmit queue. However, this will increase the queueing delays that a packet might experience. Our experiments show that increasing the transmit queue up to a point slightly decreases response time, but beyond that actually hurts response time.

6.2 Why are long requests not hurt?

In this section we give some intuition for why, in the case of web workloads, SRPT does not unfairly penalize large requests.

To understand what happens under transient overload, first consider the overload period. Under FAIR, all jobs suffer during the overload period. Under SRPT all jobs of size $< x$ complete and all other jobs receive no service, where x is defined such that the load comprised of jobs of size $< x$ equals 1. While it is true that jobs of size $> x$ receive no service under SRPT during the overload period, they also receive negligibly-little service under FAIR during the overload period because the number of connections with which they must share under FAIR increases so quickly (see Figure 6). Next consider the low load period. At start of low load, there are many more jobs present under FAIR; only large jobs are present under SRPT. These large jobs have received zero service under SRPT and negligibly-little service under FAIR until now. Thus the large jobs actually finish at about the same time under SRPT and FAIR.

The above effects are accentuated under heavy-tailed request sizes for two reasons: (1) Under heavy-tailed workloads, a very small fraction of requests make up half the load, and therefore, the fraction of large jobs ($> x$) receiving no service during overload under SRPT is very small. (2) The little service that the large jobs receive during overload under FAIR is even less significant because the large jobs are so large that, proportionately, the service appears small.

6.3 Theoretical validation

As a final step in understanding the performance of FAIR vs. SRPT, we consider an M/GI/1 queue with alternating periods of overload and low load and derive an approximation on the expected response time as a function of request size for this model under FAIR and SRPT. The derivation is too involved to include herein, but the interested reader should look at [9] for full details. We choose to evaluate our results when the service requirement distribution, G, is a Bounded-Pareto distribution with α-parameter of 1.1, as has been shown to be representative of web workloads [11].

Although the M/GI/1 queue is at best a rough approximation to our implementation setup, we nevertheless find the same trends in analysis as we have witnessed in this paper. In particular we find that the buildup in the number of requests is much greater under FAIR than under SRPT, and consequently response times are also far higher under FAIR than under SRPT. Likewise we find that

the server "recuperates" more slowly when load is dropped under FAIR than under SRPT. Also similarly to our experiments, we find that in analysis the very largest requests see approximately the same mean response time under SRPT as compared with FAIR.

7 Related Work

Our work is new in two different ways: it is the first to apply connection scheduling to improve performance of web servers under overload; and it is the first to provide an evaluation of the effect of external factors (Table 2) on the performance of an overloaded web server.

Below we describe the related work, grouped into three different categories: (1) work on improving the performance of (*non-overloaded*) web servers; since this work is numerous, we limit the description to scheduling-related work. (2) work on improving the performance of web servers under overload; (3) studies of web server performance and the effect of external factors.

Scheduling solutions – non-overloaded case

Below we summarize related work on scheduling in web servers. It is important to note that none of this prior work deals with overloaded servers. Moreover, most previous work uses simulation rather than a real implementation.

The only work that implements scheduling for web servers, rather than simulating or analytically evaluating it, is our own work [21, 31] and recent work by Rawat et. al. [54]. In [21] we experiment with connection scheduling at the *application level*. Our experimental web server improves mean response times, but at the cost of a drop in throughput by a factor of almost 2. The problem is that application level scheduling does not provide fine enough control over the order in which packets enter the network. In [31] we implement connection scheduling at the kernel-level. This eliminates the drop in throughput and offers much larger performance improvements than [21].

The authors in [54] extend the work in [31] by proposing and implementing a scheduling algorithm, which takes, in addition to the size of the request, the distance of the client from the server into account. They show that this new policy can improve the performance of large-sized files by 2.5 to 10%.

In addition to implementation work, there are simulation studies of scheduling algorithms for web servers [28,47]. Gong and Williamson [28] identify two different types of unfairness: endogenous

unfairness, that a job may suffer because of its own size, and exogenous unfairness, that a job suffers as a consequence of the other jobs it sees when it arrives. They then proceed to evaluate SRPT and other policies with respect to these types of unfairness. Murta and Corlassoli [47] develop and simulate an extension to SRPT scheduling called FCF (Fastest Connection First). Similar to [54], FCF considers WAN conditions in addition to request size when making scheduling decisions.

Finally, Friedman et. al. [27] suggest a new protocol called FSP (Fair Sojourn Protocol) for use in web servers. They show through analysis and simulation that FSP always outperforms processor sharing. Their simulation results suggest that FSP performs better than SRPT for large requests, while the opposite is true for small requests. However, they are not able to quantify these effects analytically.

Our work is different from all the above in that it focuses on the behavior of web servers under overload and is the first to show that size-based scheduling can be applied to overloaded servers without overly penalizing large requests.

Solutions for web servers under overload

There is a large body of work on systems to support high traffic web sites. Most of this work focuses on improving performance by reducing the load on the overloaded devices in the system. This is typically done in one of five ways: increasing the capacity of the system for example by using server farms or multiprocessor machines [20, 22]; using caches either on the client or on the server side [15, 17, 29]; designing more efficient software both at the OS level [8, 23, 36, 46] and the application level [51], admission control [18, 34, 61, 62, 64] or deployment of content distribution networks [35, 38, 43]. Other means of avoiding overload are content adaptation [1] and offloading work to the client [3].

Our work differs significantly from all these approaches in that we do not attempt to reduce the load on the overloaded device. Rather, our goal is to improve the performance of the system while it is overloaded. To accomplish this goal, we employ SRPT scheduling.

Studies of factors affecting web server performance

While there exist relatively few studies on servers running under overload, there are many studies of web server performance in general (not overloaded). These studies typically concentrate on the

effect that network protocols/conditions have on web server performance. We list just a few below:

In [5] and [16] the authors find that the TCP RTO value has a large impact on server performance under FreeBSD. This agrees with our study.

In [48] the authors study the effect of WAN conditions, and find that losses and delays can affect response times. They use a different workload from ours (Surge workload) but have similar findings.

The benefits of persistent connections are evaluated by [44] and [12] in a LAN environment.

There are also several papers which study real web servers in action, rather than a controlled lab setting, e.g., [45] and [58].

The work by Banga and Druschel [7] studies a web server under overload, but the only external factor considered are WAN delays. Moreover, this work focuses only on the capacity of the web server, i.e. the maximum sustainable throughput, and does not evaluate user experience or performance under fluctuating load.

8 Conclusion

The purpose of this paper is to demonstrate the effectiveness of SRPT connection scheduling in improving the performance of web servers during transient periods of overload.

We implement SRPT in an Apache web server running on Linux by scheduling the bandwidth on the server's uplink. This is done by modifying the order that socket buffers are drained within the kernel. We find that SRPT significantly improves both server stability and client experience under persistent as well as under transient overload conditions. Under persistent overload, the number of connections at the FAIR server grows quickly compared with the buildup of connections under SRPT, and consequently the FAIR server is also quick to reach the point where incoming SYNs are dropped. As a result, the client experience in terms of mean response time and the time until the first byte is received, is greatly improved under the SRPT server compared to the FAIR server. With respect to transient overload, we find that SRPT improves mean response times by factors of 1.5 – 8 over the traditional FAIR scheduling, across ten different transient overload workloads. This is significant since mean response times under FAIR can get quite high, and peak response time are many-fold higher than mean response times.

Performance improvements are measured under a vast range of environmental factors including a range of RTT's, loss rates, RTO's, persistent connections, user behaviors, packet sizes, and web server

configurations. Results are consistent across different traces. Importantly, we find that requests for large files are *not* penalized by SRPT scheduling under transient overload. In fact, for the largest 1% of requests, the response time under SRPT is very close to that under FAIR.

We conclude by noting the broader general applicability of this work. First, the approximation of SRPT implemented in this paper actually covers an entire family of algorithms, rather than one particular algorithm. The kernel-level implementation we have chosen is limited to a fixed number of priority classes, where we pick the size cutoffs between the different priority classes so as to approximate SRPT as closely as possible. Instead one could also use more conservative cutoffs and/or fewer priority classes to achieve performance closer to a FAIR system, including better performance for very large requests.

Second, while this paper focuses on static web workloads, where the bottleneck resource to be scheduled is the bandwidth on the uplink, we believe that the basic principles will extend to many other scenarios. We are currently investigating scheduling of web server backends serving *dynamic content* involving accesses to a database backend. Here, our results (see [42]) show that for database management systems (DBMS) with 2-phase locking, a transaction's life is dominated by the time spent waiting in the internal database lock queues. Thus, rather than scheduling bandwidth, as we have in this paper, for applications with dynamic content, we instead apply prioritized scheduling policies at the internal DBMS lock queues. However many of the same ideas from this paper apply.

The web is in a perpetual state of evolution with new techniques for improving performance being developed every day. When proposing a new solution for reducing web server response times one therefore needs to consider how this solution may interact with other existing or future techniques. While we have evaluated the effect of many parameters on the effectiveness of our proposed solution, we have not been able to evaluate all parameters. In particular, the effect of caches or CDNs, are left for future work.

We believe that the scheduling ideas presented in this work are general enough to be used in conjunction with other methods, or incorporated into systems other than web servers. We have already witnessed in this paper how SRPT scheduling can be used in conjunction with persistent connections. Likewise, it is likely that SRPT-like scheduling can be used in conjunction with caching schemes at a proxy server, or can be used at an edge router. For example, Biersack et al. [53] have recently proposed Least-Attained-Service (LAS) scheduling for routers, where LAS is a variant of

SRPT scheduling that doesn't require a priori knowledge of the service demand.

9 Acknowledgements

Thanks to Srini Seshan for help with porting dummynet to Linux; to Christos Gkantsidis for providing detailed information on `libwww`; to Mukesh Agrawal for helping with initial overload experiments; and to Jennifer Rexford, John Wilkes, and Erich Nahum for proofreading the paper.

References

[1] T. F. Abdelzaher and N. T. Bhatti. Web content adaptation to improve server overload behavior. *WWW8 / Computer Networks*, 31(11-16):1563–1577, 1999.

[2] W. Almesberger. Linux network traffic control — implementation overview. Available at http://lrcwww.epfl.ch/linux-diffserv/.

[3] D. Andresen and T. Yang. Multiprocessor scheduling with client resources to improve the response time of WWW applications. In *International Conference on Supercomputing*, pages 92–99, 1997.

[4] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, 2000.

[5] M. Aron and P. Druschel. TCP implementation enhancements for improving webserver performance. Technical Report TR99-335, Rice University, 6, 1999.

[6] Akamai Technologies B. Maggs, Vice President of Research. Personal communication., 2001.

[7] G. Banga and P. Druschel. Measuring the capacity of a web server under realistic loads. *World Wide Web*, 2(1-2):69–83, 1999.

[8] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of OSDI '99*, pages 45–58, 1999.

[9] N. Bansal and M. Harchol-Balter. Scheduling solutions for coping with transient overload. Technical Report CMU-CS-01-134, Carnegie Mellon University, May 2001.

[10] Nikhil Bansal and Mor Harchol-Balter. Analysis of SRPT scheduling: Investigating unfairness. In *Proceedings of* ACM SIGMETRICS '01, 2001.

[11] P. Barford and M. E. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of SIGMETRICS '98*, pages 151–160, July 1998.

[12] P. Barford and M. E. Crovella. A performance evaluation of hyper text transfer protocols. In *Proceedings of ACM SIGMETRICS '99*, pages 188–179, May 1999.

[13] M. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[14] D. J. Bernstein. Syn cookies. http://cr.yp.to/syncookies.html, 1997.

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

[16] L. Brakmo and L. Peterson. Performance problems in 4.4 BSD TCP. *ACM Computer Communications Review*, 25(5), 1995.

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

[18] L. Cherkasova and P. Phaal. Session based admission control: A mechanism for improving the performance of an overloaded web server. Technical Report HPL-98-119, Hewlett Packard Laboratories, 6 1998.

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

[20] M. Colajanni, P. S. Yu, and D. M. Dias. Analysis of task assignment policies in scalable distributed Web-server systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(6):585–699, 1998.

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

[22] D. M. Dias, William Kish, Rajat Mukherjee, and Renu Tewari. A scalable and highly available web server. In *COMPCON*, pages 85–92, 1996.

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

[24] Sameh Elnikety, Erich M. Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in dynamic e-commerce Web sites. In *International World-Wide Web Conference (WWW)*, New York, NY, May 2004.

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

[26] Cooperative Association for Internet Data Analysis (CAIDA). Packet length distributions. http://www.caida.org/analysis/AIX/plen_hist, 1999.

[27] E. J. Friedman and S. G. Henderson. Fairness and efficiency in web server protocols. In *Proceedings of the 2003 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 2003.

[28] Mingwei Gong and Carey Williamson. Quantifying the properties of SRPT scheduling. In *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2003.

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

[30] Internet Town Hall. The internet traffic archives. Available at `http://town.hall.org/Archives/pub-/ITA/`.

[31] M. Harchol-Balter, B. Schroeder, M. Agrawal, and N. Bansal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems*, 21(2), May 2003.

[32] IRCache Home. The trace files. http://www.ircache.net/Traces/, 2004.

[33] Microsoft TechNet Insights and Answers for IT Professionals. The arts and science of web server tuning with internet information services 5.0. http://www.microsoft.com/technet/, 2001.

[34] R. Iyer, V. Tewari, and K. Kant. Overload control mechanisms for web servers. In *Workshop on Performance and QoS of Next Generation Networks*, November 2000.

[35] Kirk L. Johnson, John F. Carr, Mark S. Day, and M. Frans Kaashoek. The measured performance of content distribution networks. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, 2000.

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

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

[38] B. Krishnamurthy, C. Wills, and Y. Zhang. The use and performance of content distribution networks. In *Internet Measurement Workshop. Proceedings of the First ACM SIGCOMM Workshop on Internet Measurement Workshop.*, 2001.

[39] J. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet.* Addison Wesley Longman, Inc., 2001, pp.319.

[40] W. LeFebvre. Cnn.com: Facing a world crisis. Invited talk at the USENIX Technical Conference, June 2002.

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

[42] D. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *20th International Conference on Data Engineering (ICDE 2004)*, 2004.

[43] M.Day, B.Cain, G.Tomlinson, and P.Rzewski. A model for content internetworking (cdi). Internet Draft (draft-ietf-cdi-model-02.txt), May 2002.

[44] J. C. Mogul. The case for persistent-connection HTTP. In *Proceedings of ACM SIGCOMM '95*, pages 299–313, October 1995.

[45] J. C. Mogul. Network behavior of a busy Web server and its clients. Technical Report 95/5, Digital Western Research Laboratory, October 1995.

[46] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. USENIX 1996 Technical Conference*, pages 99–111, 1996.

[47] C.D. Murta and T.P. Corlassoli. Fastest connection first: A new scheduling policy for web servers. In *Proceedings of the 18th International Teletraffic Congress (ITC-18)*, September 2003.

[48] E. Nahum, M. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on www server performance. In *Proceedings of* ACM SIGMETRICS '01, pages 257–267, 2001.

[49] National Institute of Standards and Technology. Nistnet. `http://snad.ncsl.nist.gov/itg/nistnet/`.

[50] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose. Modeling tcp reno performance: a simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8(2):133–145, 2000.

[51] Vivek S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of USENIX 1999*, June 1999.

[52] V. Paxson and M. Allman. Computing TCP's retransmission timer. RFC 2988, `http://www.faqs.org/rfcs/rfc2988.html`, November 2000.

[53] Idris A. Rai, Guillaume Urvoy-Keller, and Ernst Biersack. Analysis of LAS scheduling for job size distributions with high variance. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2003.

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

[55] Internet Traffic Report. http://www.internettrafficreport.com, 2004.

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

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

[58] S. Seshan, Hari Balakrishnan, V.N. Padmanabhan, M. Stemm, and R. Katz. TCP behavior of a busy internet server: Analysis and improvements. In *Proceedings of* Conference on Computer Communications (IEEE Infocom), pages 252–262, 1998.

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

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

[61] T. Voigt and P. Gunnigberg. Kernel-based control of persistent web server connections. *ACM SIGMETRICS Performance Evaluation Review*, 29(2):20–25, 2001.

[62] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[63] The World Wide Web Consortium (W3C). Libwww - the W3C protocol library. `http://www.w3.org`.

[64] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *Proceedings of the 2003 USENIX Symposium on Internet Technologies and Systems*, 2003.

[65] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an M/GI/1. In *ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, 2003.

Appendix: Another Trace

In this appendix we consider one more trace and run all experiments on the new trace. We find that the results are very similar to those shown in the body of the paper, both with respect to comparative mean response times for the different scenarios (A) through (J), and with respect to unfairness issues.

The log used here was collected from a NASA web server and is also available through the Internet Traffic Archive [30]. We use several hours of one busy day of this log consisting of around 100000 mostly static requests. The minimum file size is 50 bytes, the maximum file size is 1.93 Mbytes. The largest 2.5% of all requests make up 50% of the total load, exhibiting a strong heavy-tailed property. The primary statistical difference between the NASA log and the soccer World Cup log (used in body of the paper) is the mean request size: the NASA log shows a mean file size of 19 Kbytes while for the World Cup log it was only around 5 Kbytes.

Results for the NASA log are shown in Figure 13, 14 and 15. They are extremely similar to the corresponding Figures 9, 10, 11 and 12 for the World Cup trace.

The only difference is that response times are higher under the NASA log as compared with the World Cup log for both FAIR and SRPT. The relative performance gains of SRPT and FAIR are similar. The increase in response times under the NASA log may be attributed to the higher mean file size.

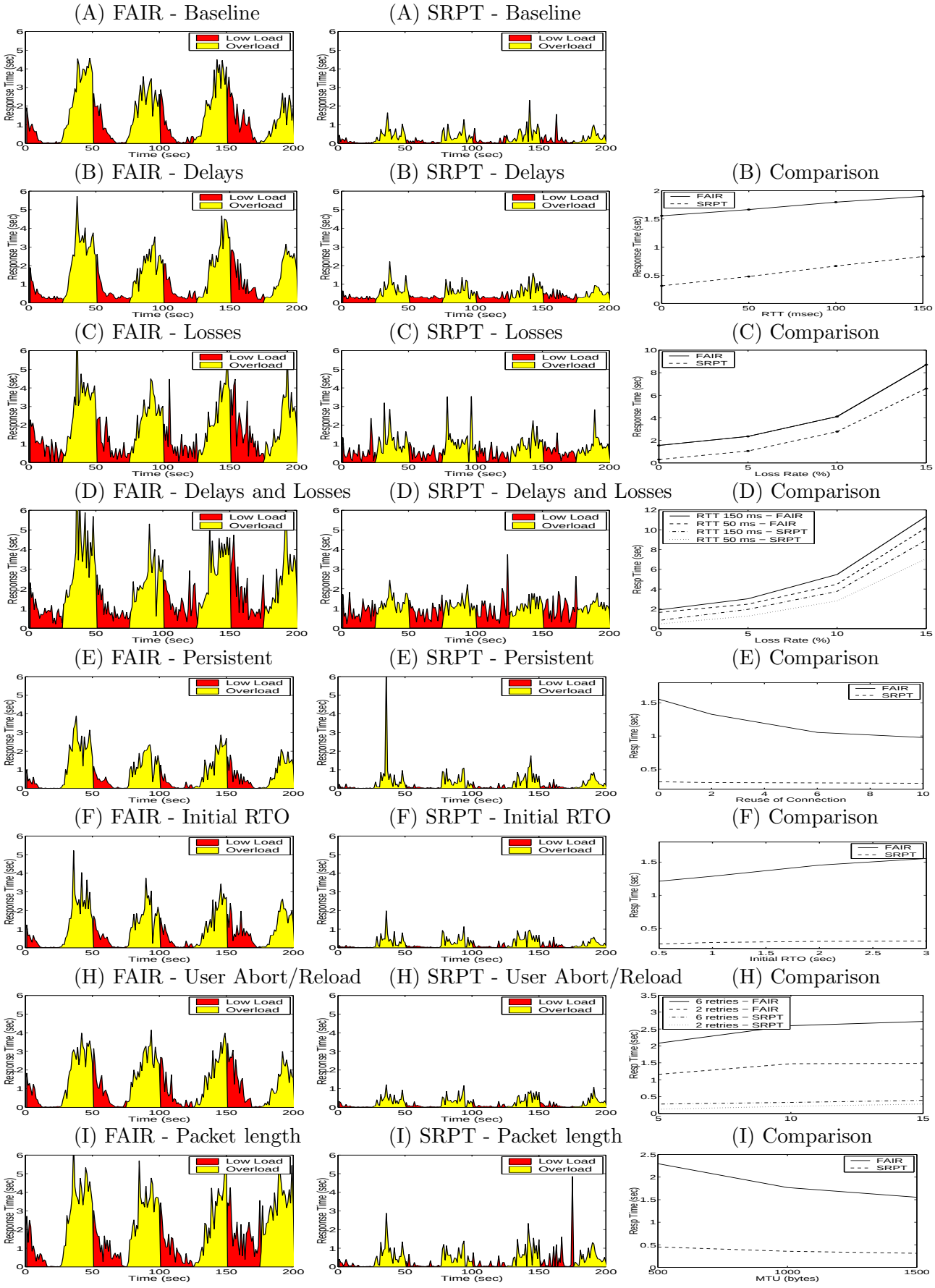
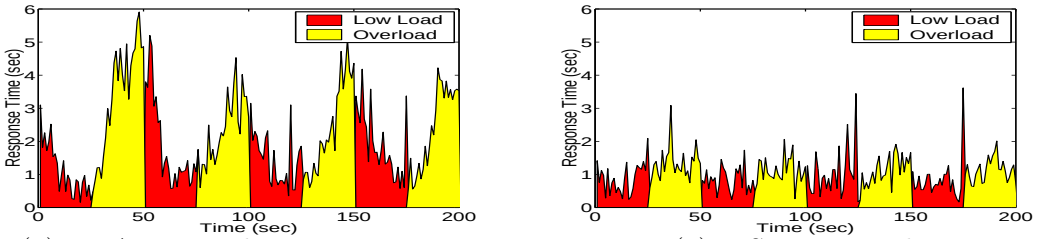


Figure 13: *Results under NASA trace log. Each row in the figure compares SRPT and FAIR under workload W1 for one particular setup from Table 2. The left and middle columns show the response times over time for the specific values given in Table 2, column 3. The right column evaluate the range of values given in the column 4 of Table 2.*



(J) FAIR - Realistic case (J) SRPT - Realistic case

Figure 14: *Comparison of FAIR (left) and SRPT (right) for the realistic setup for workload W1, under NASA trace log.*

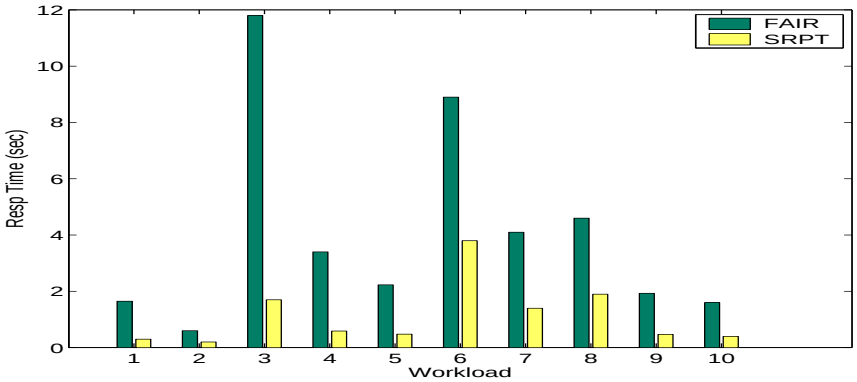


Figure 15: *Mean response time comparison of FAIR and SRPT in the baseline case for the workloads in Table 1, under the NASA trace log.*