Web servers under overload: How scheduling can help

Bianca Schroeder Mor Harchol-Balter[1]

May 2002
CMU-CS-02-143

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Most well-managed web servers perform well most of the time. Occasionally, however, every popular web server experiences transient *overload*. 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, causing the number of connections at the server to build up. This implies large delays for clients accessing the server.

This paper provides a systematic performance study of exactly what happens when a web server is run under transient overload, both from the perspective of the server and from the perspective of the client. Second, this paper proposes and evaluates a particular kernel-level solution for improving the performance of web servers under overload. The solution is based on SRPT connection scheduling.

We show that SRPT-based scheduling improves overload performance across a variety of client and server-oriented metrics.

[1]This work was supported by NSF Career Grant CCR-0133077 and by Spinnaker Networks through Pittsburgh Digital Greenhouse Grant 01-1.

Keywords: Web servers, overload, transient behavior, connection-scheduling, HTTP requests, priority, M/G/1, shortest processing remaining time, starvation, heavy-tailed workloads, time-sharing, Linux, Apache.

1 Introduction

"A web site that fails to deliver its content, either in a timely manner or at all, causes visitors to quickly lose interest, wasting the time and money spent on the site's development"[43].

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 due to the burstiness of web traffic and web hot spots.

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 which 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 contribution of this paper is two-fold: First, the paper provides a performance study of a web server under overload. We experiment with both *persistent overload* (load exceeds 1 during the entire experiment) and *transient overload* (alternate between periods of overload and low load). We evaluate a full range of complex environmental conditions, which are summarized in Table 2. These include: 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, the effect of the SYN queue and ACK queue lengths. Second, the paper proposes and evaluates a particular kernel-level improvement based on connection scheduling.

Our work is based on a standard Apache web server running on Linux, and servicing *static* requests, of the form "Get me a file." The workload in all experiments is based on web traces. Measurements [31, 30, 24] have suggested that the request stream at most web servers is dominated by *static* requests. Serving static requests *quickly* is the focus of many companies *e.g.*, Akamai Technologies, and much ongoing research.

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 which the server has bought from its ISP [28, 18, 5]. 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 fact, most web sites buy sufficient memory so that all their files fit within memory (keeping disk utilization low) [5]. For static workloads, CPU load is typically not an issue.

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 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 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.

To evaluate the performance of our server under overload, we use a trace-based workload and generate requests according to an open system model as shown in Figure 4. We evaluate performance both from the client and the server perspective. To evaluate the client experience we record at the client side *response time*. This is the time from when a client first sends out the SYN until the client receives the last byte. To monitor the server's health we have instrumented the kernel in the server to provide us with detailed information about the state of the server such as the length of the SYN-queue, the length of the ACK queue, number of active connections and other statistics. Under all conditions tested, we find that under even mild overload, the number of connections grows rapidly, resulting in very high response times.

These high response times motivate us to look at new ways to reduce the *queueing delays* at web servers under overload using connection scheduling. 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 short requests, or those requests with short remaining time, in accordance with the well-known scheduling

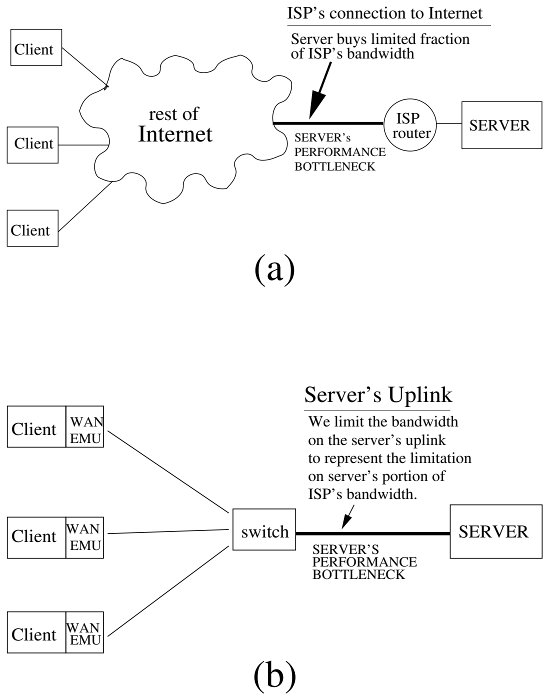


Figure 1: *(a) Server's bottleneck is the limited fraction of bandwidth that it has purchased from its ISP. (b) How our implementation setup models this bottleneck by limiting the server's uplink bandwidth.*

algorithm preemptive Shortest-Remaining-Processing-Time-first (**SRPT**).

Although it is well-known from queueing theory that SRPT scheduling minimizes queueing time [40], applications have shied away from using SRPT for two reasons: First SRPT requires knowing the *size of the request*[1] (i.e. the time required to service the request). Our experiments show that the size of a request is well-approximated by the file size (modulo a small overhead), which is well-known to the server. Second, there is the fear that SRPT "starves" big requests [11, 19, 42]. The intuition commonly given is that a big request won't complete because it will continuously be interrupted by the arrival of more and more small requests. The above statement would be obvious if the system were permanently overloaded. However, it's not clear what happens in the case of *transient overload*. A major goal of this paper is to resolve this question.

Since the server's uplink link is the bottleneck resource, we apply the SRPT idea to the uplink. Our idea is to control at the kernel level the order in which the server's socket buffers are drained onto the link. Traditionally, the different socket buffers are drained in Round-Robin Order (each getting a fair share of the bandwidth on the uplink). We call such a traditional server the **FAIR** server. We instead propose modifying the server to give priority to those sockets corresponding to connections for small file requests or those for which the *remaining data* required is small. We refer to this modified server as the **SRPT** server.

1.1 Organization of paper

Section 2 explains our implementation of SRPT. Section 3 describes the experimental setup used throughout the paper. Section 4 studies exactly what happens in a traditional (FAIR) web server under overload and contrasts that with the performance of the modified (SRPT) web server. Section 5 analyzes where exactly SRPT's performance gains come from. Section 6 discusses relevant previous work, and Section 7 concludes.

2 Implementing an SRPT web server

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

2.1 Achieving priority queueing in Linux

Figure 2 (left) shows the data flow in standard Linux.

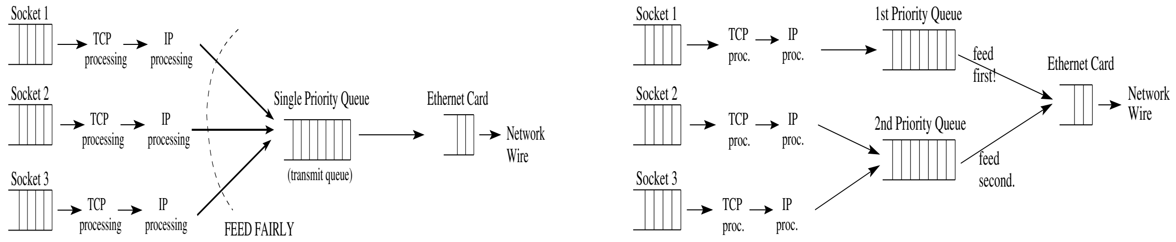
There is a socket buffer corresponding to each connection. Data streaming into each socket buffer is then processed by TCP and IP. Throughout this processing, the packet stream corresponding to each connection is kept separate from every other connection. Finally, there is a *single*[2] "priority queue", into which *all* streams feed. Provided that all streams are identical (have equal amounts of data ready to send), they get equal turns draining into the priority queue. 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. Our approach is to 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`[2] user space tool to switch the device queue from the default 3-band queue to the 16-band prio queue.

Figure 2 (right) shows the flow of data in Linux after the above modification. Again, the data is passed from the socket buffers through TCP and IP processing. Throughout this processing, the packet streams corresponding to each connection are kept separate. Finally,

[1] Strictly speaking, it is not the size of the request but the size of the response that we are talking about. We use these two terms interchangeably.

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



(a) Standard Linux - FAIR

(b) Modified Linux - SRPT

Figure 2: *Data flow in standard Linux (left) and in Linux with priority queueing (right).*

there are 16 priority queues. 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. Observe that since scheduling occurs after TCP processing it does not interfere with TCP or IP.

2.2 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. 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 somewhat important with respect to the performance of the web server. We have experimentally derived some good *rules-of-thumb* 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. The intuition here is that the smallest 50% of requests comprise so little total load in a heavy-tailed distribution that there's no point in separating them.
- 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. Anything remotely close to a logarithmic spacing works well.

In the experiments throughout this paper, we use only 5 priority classes (1KB, 2KB, 5KB, and 50KB) to approximate SRPT. Using more improved performance only slightly.

A potential problem with our approach is the overhead of the `setsockopt` system call 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.

3 Experimental Setup

We start with a brief overview over some important server internals in Section 3.1. Section 3.2 describes our machine configuration; Section 3.3 defines the trace-based workload that we use and the request generator at the clients. Finally, Section 3.4 describes how we emulate WAN conditions in our setup.

3.1 Background on Server Internals

For a better understanding of our choice of performance metrics and of the results of our performance study, we review the processing of an incoming request in an Apache server running on Linux.

A connection begins with a client sending a `SYN`. When the server receives the `SYN` it allocates 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. Apache maintains some number of ready processes, each of which 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` is 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). Both the size of the queues and the number of Apache processes is 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.

3.2 Machine Configuration

Our experimental setup involves six machines connected by a 10/100 Ethernet switch. 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 in Section 3.3. 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[3].

On the server machine we increased the size of the SYN and the ACK-queue as is common practice for high performance web servers We experimented with several values for the size of these queues and found a size of 512 for each of them to give the best results for our system. We also increased the upper limit on the number of Apache processes from 150 to 350.[4].

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.

[3]Experiments were also performed under 100 Mbps mode, but not in overload since that puts too much strain on the client machines, see Section 3.3.3.

[4]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.

3.3 Workload

3.3.1 The trace

The workload is based on a 1-day trace from the Soccer World Cup 1998 obtained from the Internet Traffic Archive [27]. The trace contains 4.5 million mostly static HTTP requests. In our experiments, we use the trace to specify the time at which the client makes the request and the request size in bytes. Results for experiments with other logs are given in Appendix B.

Some statistics about our trace workload: The mean file size requested is 5K bytes. The min size file requested is a 41 byte file. The max size file requested is a 2.02 MB file. The distribution of the file sizes requested fits a *heavy-tailed* Pareto distribution. The largest $< 3\%$ of the requests 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. These characteristics are similar to that of workload generated by Surge[9].

3.3.2 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. However, if a real web server is persistently overloaded the inevitable consequence is to upgrade the system. 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 type, called *intermittent overload*, the load is almost always low, but there are occasional "spikes" of overload, evenly spaced apart (see Figure 3(right)).

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 appropriately.

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 min experiments although we show only shorter fragments in the figures for better readability.

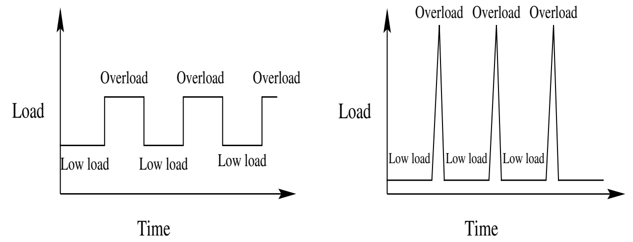


Figure 3: *Two different types of transient overload, alternating (left) and intermittent (right). Note, that in the experiment, the load will never look as constant as in the figure, since arrival times and request sizes come from a trace.*

Work-load	Type	Duration low load	Duration overload	Low load	Over-load	Avg. load
W1	Alt	25 sec	25 sec	$\rho = 0.2$	$\rho = 1.2$	0.7
W2	Alt	10 sec	10 sec	$\rho = 0.2$	$\rho = 1.2$	0.7
W3	Alt	50 sec	50 sec	$\rho = 0.2$	$\rho = 1.2$	0.7
W4	Alt	25 sec	25 sec	$\rho = 0.4$	$\rho = 1.4$	0.9
W5	Alt	10 sec	10 sec	$\rho = 0.4$	$\rho = 1.4$	0.9
W6	Int	20 sec	3 sec	$\rho = 0.5$	$\rho = 2$	0.7
W7	Int	13.3 sec	2 sec	$\rho = 0.5$	$\rho = 2$	0.7
W8	Int	20 sec	3 sec	$\rho = 0.735$	$\rho = 2$	0.9
W9	Int	13.3 sec	2 sec	$\rho = 0.735$	$\rho = 2$	0.9

Table 1: Definition of workloads

3.3.3 Generating requests at the client machines

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 which the server has repeatedly refused (due to full SYN-queue).

Existing web workload generators include `Surge` [9] and `Sclient` [6]. 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). Although Surge was easily modified into an open system, we found that it does not scale to the high number of connections that our clients need to maintain. This is because Surge is based on *multithreading*, maintaining one thread per connection.

Using Sclient (based on the *select()* call) to handle multiple connections simultaneously, we were able to generate the required load. However, Sclient does not support persistent connections.

Instead of extending Sclient's functionality, we chose to implement our own trace-based web workload generator based on the `libwww` library [45]. `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 modified `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.

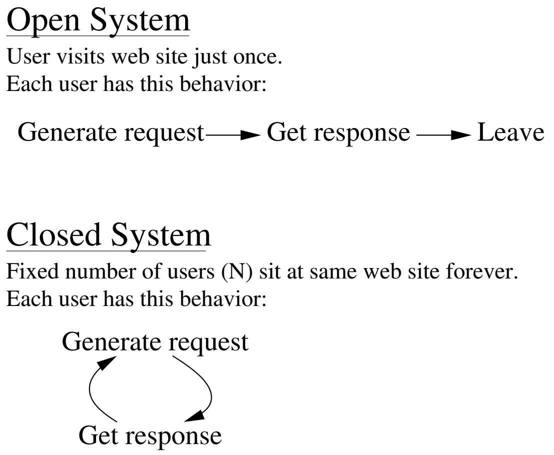


Figure 4: *Two models for how the requests to a web server are generated. In creating overload, one must use an open system model.*

We ran all the experiments that do not involve persistent connections with both Sclient and our new workload generator and found the results to be the same.

3.4 Emulating WAN effects

The two most frequently used tools for WAN emulation are probably NistNet [36] and Dummynet [39].

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 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 transmission of delayed packets. We recompile the kernel with `HZ=1000` to get a finergrained 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.

4 Experimental Results

In this section we study exactly what happens in a standard (FAIR) web server under overload and contrast that with the performance of our modified (SRPT) server. Table 2 describes the various factors we consider in our study. In Section 4.1, we evaluate performance for workload W1 only. In Section 4.2, we consider the other workloads in Table 1.

4.1 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 look at *persistent* overload and then at *transient* overload. Since the main observations for persistent overload are the same as for the overload portion of transient overload, we later consider only transient overload.

Persistent overload

In this section 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 5(left). In FAIR, after 40 to 50 sec 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 and incoming `SYN`s are dropped. In the SRPT server the number of connections grows at a much slower rate. The growth rate under SRPT is imperceptible under the short timescale in the figure – it takes more than 300 seconds until the maximum number of connections is reached. Once the SRPT and the FAIR server start to drop `SYN`s they drop `SYN`s at a comparable rate.

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 computing the mean response time for persistent overload, we consider only those requests that finished before the last request arrived (subsequently, load will drop). We observe that for both servers the mean response times increase steadily over time (see Figure 5(right)). In the FAIR server at time 10 sec (long before the SYN-queue fills up) the mean response time is 5 sec, already intolerable. Under SRPT the response times grow at a much slower rate. Even after 70 sec the response time under SRPT is less than 5 sec.

To summarize the above observations, we saw that after less than 50 seconds of moderate 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 `SYN`s are dropped and improves the client experience notably.

Transient Overload

In this section we use the transient workload example W1 introduced in Table 1 to study the performance under transient overload.

We first consider how the health of the server is affected during the low load and overload periods. We observe again 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 `SYN`s are dropped.

Figure 6 (A) shows the response times over time of all jobs (averaged over 1 sec intervals). What is shown in all these plots is just 200 sec out of a 30-minute experiment. The overload periods are shaded light grey while the low load periods are shaded dark grey. 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 7(left) shows the distribution of the response times under FAIR and SRPT. 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 compared to only 138 ms under SRPT. Furthermore, the variance 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

[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 affecting performance	Specific case shown in Figure 6(left) and (middle)	Range of values studied and shown in Figure 6(right)
(A)	Baseline Case	RTT=0, Loss=0%, No Persistent Conn., RTO=3 sec, pkt. len.=1500, No SYN cookies, SYNQ=ACKQ=512, #ApacheProcs=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 6(left, middle). Column 4 provides a range of values for each factor. The range is evaluated in Figure 6(right)).*

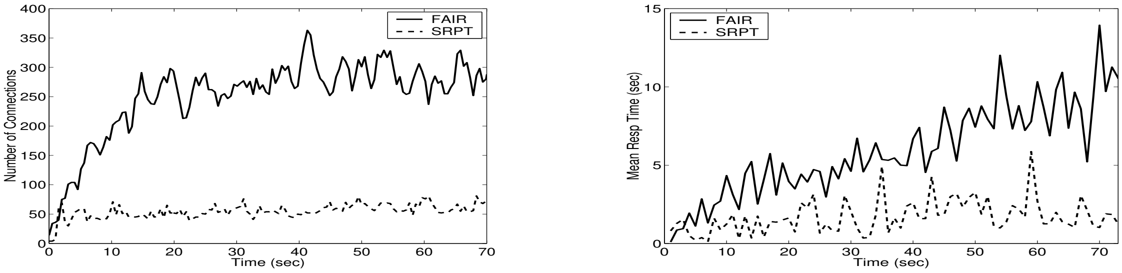


Figure 5: *Results for a persistent overload of 1.2. (Left) Buildup in connections at the server. (Right) response times.*

loss during connection setup. Given that we have a virtually loss-free LAN setup and the SYN-queue never fills up, the question is 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 5.

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 7(right) 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.

Even for the very biggest request the mean response time is 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 5.

(B) WAN delays

Figure 6(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 6(B)(right) shows the mean response times for various RTTs ranging 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.

(C) Network losses

Figure 6(C)(left, middle) shows the mean response times over time for the setup in Table 2 row (C): a loss rate of 5%. Note that 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 sec to 1.9 sec. SRPT's response time increases from less than 140 ms in the baseline case to around 930 ms.

Figure 6(C)(right) shows the mean response times for loss rates ranging from 0 to 15%. We observe that the response times don't grow linearly in the loss rate. This

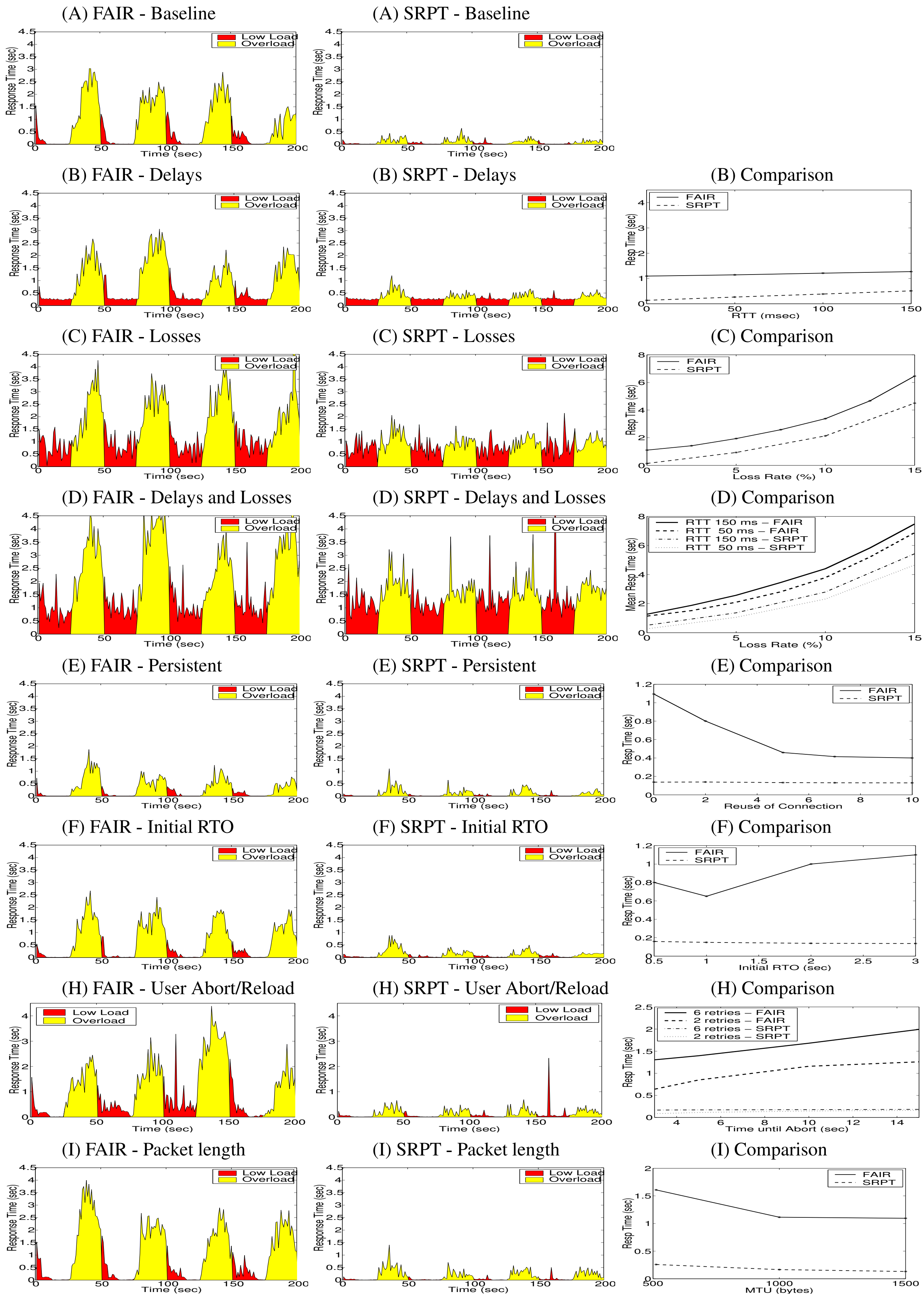


Figure 6: *Each row in the figure compares SRPT and FAIR under workload W1 for one particular setup from Table 2. The left and middle column in the figure show the response times over time for the specific values given in Table 2, column 3. The right column in the figure compares the performance of SRPT and FAIR for the range of values given in the column 4 of Table 2.*

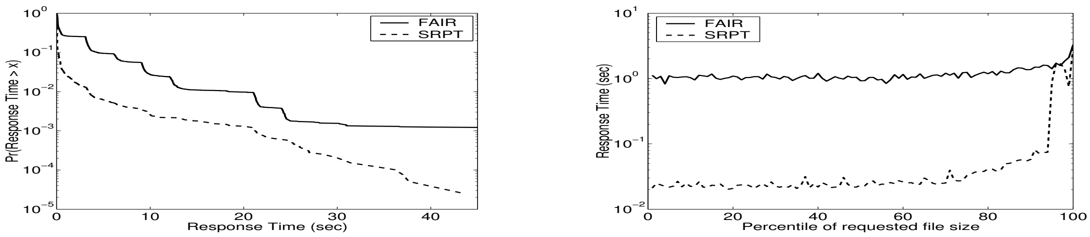
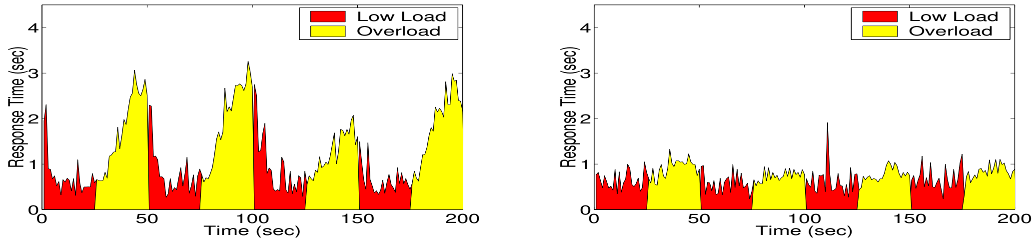


Figure 7: *The distribution of response times (left) and the response times as a function of the request size (right) for W1.*



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

Figure 8: *Comparison of FAIR (left) and SRPT (right) for the realistic setup for workload W1.*

is to be expected since TCPs throughput is inversely proportional to the square root of the loss.

Introducing frequent losses can increase FAIR's response times to more than 6 sec and SRPT's response time to more than 4 sec (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 6(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 6(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 slightly bigger if the loss rate is higher. 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 ([32]). for several requests. Figure 6(E)(left, middle) shows the results for the setup in Table 2, row (E): where every connection is reused 5 times. Figure 6(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. 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 changed the RTO value in Linux's TCP implementation. Figure 6(F)(left, middle) shows the results for the setup in Table 2, row (F): an RTO of 500ms. Figure 6(F)(right) explores the range from 500 ms up to the standard 3 sec. We observe that by setting the initial RTO to 500 ms, we can reduce FAIR's mean response time from 1.1 sec to 0.8 sec. 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 there is always at least a factor of 4 improvement under SRPT as compared to FAIR.

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 [38].

(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 [12] 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. Our hope is by getting rid of the SYN-queue to also get rid of the problems involving a full SYN-queue.

The use of SYN cookies hardly affects the response times under W1 since in 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 `SYN`s 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 6(H)(left,middle) shows the response times if a user waits for at most 10 sec before he hits reload and retries this procedure up to 6 times before giving up. Figure 6(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, both increase or 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 sec and retry at most 2 times, more than 5 percent of the requests under FAIR never complete for 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, which might spend quite some time on requests that later get aborted. For example, if users retry up to 6 times and abort a connection only after 15 sec 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%.

In comparing FAIR and SRPT we observe that 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). There are two different packet lengths commonly observed in the Internet [25]: 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.

Figure 6 (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 sec and SRPT's response time doubles to 260 msec.

Figure 6 (I)(right) shows the mean response times for different packet lengths ranging from 500 bytes to 1500 bytes. For all packet lengths considered, SRPT always 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 8 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 sec 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 sec 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 sec and 1.2 sec, 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 sec under FAIR and 2.28 sec under SRPT. Even the response time of the very biggest request is higher under FAIR: 13.2 sec under FAIR and only 12.8 sec under SRPT. The total number of incomplete requests is similar 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.

4.2 Other transient workloads

Figure 9(left) gives the mean response times for all workloads from Table 2 for the baseline case (Table 2 row (A)), and Figure 9(right) shows the mean response time for the largest 1% of requests, again for the baseline case. 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

We find that for the *baseline case*, the mean response time of SRPT improves upon that of FAIR by a factor of 4 – 8 across the different workloads. More specifically, FAIR ranges from 300ms – 2.5 sec, while SRPT ranges from 50ms – 400 ms.

Under the *realistic case* SRPT improves mean response times upon that of FAIR by a factor of 1.5 – 4 across the different workloads. FAIR ranges from 900ms – 3 sec, while SRPT ranges from 600ms – 900 ms.

We find that 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. For example workloads W4, W5, W8, W9, with high mean system load, also have higher mean response times. Next consider Workload 1, 2 and 3 which differ only in the length of the overload and low period; W2, which has a 10 sec overload and low load period, leads to a 10 times lower mean response time than W3, where each low load and high load periods lasts 50 sec. The length of the overload period and the overall load also affect the number of `SYN` drops, and consequently the response times. Most workloads have zero `SYN` drops, under both the baseline and realistic setups. However workloads W3 and W4, which have longer overload periods, result in 1 – 3% `SYN` drops under the baseline case under FAIR and 20 – 23% `SYN` drops under the realistic case under FAIR. This explains the high response times of workloads W3 and W4 under FAIR. `SYN` drops do not occur under SRPT.

Performance of large requests

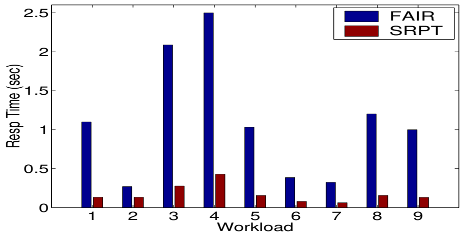
Again, an obvious question to ask is whether this improvement in the mean response time comes at the price of longer response times for the long requests. In particular, for the workloads with higher mean system load and longer overload periods it might seem likely that SRPT leads to a higher penalty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

Figure 9(right) shows for each workload the response times of only the biggest 1% of all requests in the *baseline case*. The mean response time for the top 1% of requests is never more than 10% higher under SRPT than FAIR for any workload.

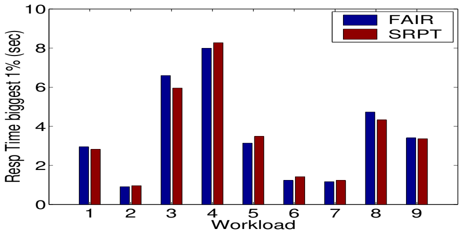
Under the *realistic setup*, the mean response time for the top 1% of requests is never more than 2% higher under SRPT than under FAIR, where we count only completed requests.

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 (since the realistic setup incorporates 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. Looking only at the largest 1% of requests, the fraction of incomplete requests is much more variable, ranging from 3 – 14% under SRPT and 3 – 26% under FAIR, but is always 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 (W3) or the mean system load (W4, W5, W8, W9)



Mean Response Time - Baseline



Mean Response Time of biggest 1% requests - Baseline

Figure 9: *Comparison of FAIR and SRPT in the baseline case for the workloads in Table 1. The mean response times over all requests are shown left, and the mean response times of only the biggest 1% of all requests are shown right.*

does *not* result in more starvation. This also agrees with the theoretical results in Appendix A.

5 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.

5.1 Where do mean gains come from?

Recall, that we identified in Section 4 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 5.

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 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 6(F). Observe that even when problem 3 is removed, the improvement of SRPT over FAIR is still a factor of 7. We therefore conclude that problem 1 is the main reason why SRPT improves upon FAIR.

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. We observed experimentally that increasing the transmit queue up to a point slightly decreases response time, but beyond that actually hurts response time.

5.2 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 5). 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.

The intuition given above is formalized using queueing-theoretic analysis in Appendix A.

6 Related Work

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 four 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 [26, 15, 13]; designing more efficient software both at the OS level [34, 7, 23, 29] and the application level [37], and admission control [17, 44]. 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. While there has been some work done on scheduling in web servers, e.g. [21, 16], no prior work focuses on size-based scheduling to improve overload 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 [4] and [14] the authors find that the TCP RTO value has a large impact on server performance under FreeBSD. This agrees with our study.

In [35] 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 [32] and [10] in a LAN environment.

There are also several papers which study real web servers in action, rather than a controlled lab setting, e.g., [33] and [41].

7 Conclusion

This paper provides a detailed performance study of the effect of overload on an Apache web server running over Linux and servicing static content. We start with a baseline setup consisting of a simple LAN and find that under even slight overload, the number of connections at the server grows rapidly, resulting in response times on the order of seconds after only a couple seconds.

We next evaluate the server under nine types of transient overload, where load oscillates between overload and low load, however the mean load is well under 1. We find that performance is dependent on the length of the overload period and also on whether the transient overload is of the alternating or intermittent type. We see that under alternating overload, mean response times are quite high – typically a second or two.

We next consider the effect of external factors such as WAN conditions (losses and delays), packet lengths, RTO timer values, SYN cookies, persistent connections, and user aborts, first in isolation and then in combination. Some factors like persistent connections improve performance, while others, like loss, greatly hurt performance. When taken in combination, under realistic settings for each of these factors, we find that mean response times are still very high.

These results motivate us to consider how scheduling of connections at the web server might be used to improve performance. In particular we consider replacing the traditional FAIR scheduling used in web servers today by SRPT-based scheduling.

We discover a *basic principle*: Scheduling to favor short requests can have a big impact on mean response time without causing starvation of long requests and without reducing throughput. We demonstrate this basic principle both experimentally and analytically.

For the baseline scenario (assuming only a simple LAN environment with no persistent connections and no user aborts) we find that SRPT improves mean response times by 200 – 800 % over the traditional FAIR scheduling, across nine transient overload workloads. We further find that for the largest 1% of requests, the response time under SRPT is no more than 10% higher than that under FAIR, for any workload.

We next consider whether the external factors mentioned above might overturn the basic principle. We find that while most of these factors (particularly WAN conditions and persistent connections) diminish the improvement factor of SRPT over FAIR, the basic principle still stands. Specifically, under a realistic scenario which combines all of the above factors, the improvement of SRPT over FAIR with respect to mean response time is still significant: ranging from 150% to 400% across nine workloads, while the largest 1% of all jobs perform only 2% worse under SRPT than under FAIR.

We conclude by mentioning that although in this paper the resource scheduled was the bandwidth on the uplink, we believe that the basic principle will extend to scenarios where the bottleneck resource is the CPU or the disk-to-memory bandwidth. Furthermore, although this paper has focused on web servers, these ideas may be applicable to other server systems as well.

8 Acknowledgements

Thanks to Mukesh Agrawal for helping us with initial overload experiments and to Nikhil Bansal for work on Appendix A. Thanks to Christos Gkantsidis for providing detailed information on `libwww`. Thanks to Srini Seshan for help with porting dummynet to Linux. Finally, thanks to Jennifer Rexford, John Wilkes, 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. Aron and P. Druschel. TCP implementation enhancements for improving webserver performance. Technical Report TR99-335, Rice University, 6, 1999.

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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] H. Chen and P. Mohapatra. Session-based overload control in qos-aware web servers. In *Proceedings of IEEE Infocomm '02*, 2002.

[17] L. Cherkasova and P. Phaal. Session based admission control: A mechanism for improving the performance of an overloaded web server. Available at http://www.hpl.hp.com/techreports/98/HPL-98-119.html., 1998.

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

[19] E.G. Coffman and L. Kleinrock. Computer scheduling methods and their countermeasures. In *AFIPS conference proceedings*, volume 32, pages 11–21, 1968.

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

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

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

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

[28] 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.

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

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

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

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

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

[34] 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.

[35] 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.

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

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

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

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

[40] L. E. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16:678–690, 1968.

[41] 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.

[42] A. Silberschatz and P. Galvin. *Operating System Concepts, 5th Edition*. John Wiley & Sons, 1998.

[43] Freshwater Software. Web server monitoring. Available at http://www.freshtech.com/white_paper/bookchapter/chapter.htm.

[44] 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.

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

9 Appendix A: Theoretical Results

As a final step in understanding the performance of FAIR vs. SRPT, we consider an M/G/1 queue with alternating periods of overload and low load and derive the expected response time as a function of request size for this model under FAIR and SRPT.

To the best of our knowledge no prior results exist for FAIR or SRPT under alternating overload/low load. The M/G/1 queue is at best a rough approximation to our implementation setup, and furthermore our theorems require some simplifying assumptions. Nevertheless, we will see that the queueing theoretical results which we obtain below are predictive of the implementation results we obtained.

The job sizes are assumed to be independent and identically distributed with c.d.f. $F(x)$ and p.d.f. $f(x)$. During the high load period (of duration t_h), jobs arrive with mean rate λ_h and create a load of $\rho_h > 1$. During the low load period (of duration t_l), jobs arrive with mean rate λ_l and create a load of $\rho_l < 1$. We denote the distribution of the remaining service requirement of requests at the start of the LOW period by F_r.

Below, we state only our theorems for the case $\rho_L = 0$. We refer to this as the ON/OFF model. Our theorems for the case $\rho_L > 0$ require more notation/definitions than we have space for, but can be found in [8].

Theorem 1 (Approximation) *In an M/G/1 system, under the above ON/OFF load model, let* $\mathbf{E}\{T(x)_{FAIR}\}$ *(respectively* $\mathbf{E}\{T(x)_{SRPT}\}$*) denote the mean response time for a request of size* x *under FAIR (respectively, SRPT). Then*

$$\begin{aligned}\mathbf{E}\{T(x)_{FAIR}\} = \quad & (\rho_h - 1)F_{r_e}(x)t_h - \tfrac{t_h}{2}(\overline{F}_r(x) - e^{-ax}) \\ & + \tfrac{(1-e^{-ax})}{2}t_h - \tfrac{\lambda_h t_h}{2}\int_0^x \overline{F}(x-z)e^{-az}dz\end{aligned}$$

where a *is the solution to the following equation:*

$$\begin{aligned}1 - \int_0^\infty f(x)e^{-ax}dx &= a/\lambda_h \text{ and} \\ \overline{F_r}(y) &= \int_y^\infty \lambda_h f(z)(1-e^{-a(z-y)})a^{-1}dz \\ F_{r_e}(x) &= a(\rho_h - 1)^{-1}\int_0^x \overline{F_r}(y)dy \qquad .\end{aligned}$$

$$\mathbf{E}\{T(x)\}_{SRPT} = o(t_h), \quad if \quad x < x_o$$

$$\mathbf{E}\{T(x)\}_{SRPT} = t_h\rho_h(x) - \frac{t_h}{2}, \quad if \quad x > x_o$$

where x_o *is such that the load made up by requests of size* $< x_o$ *is exactly* 1*, i.e.* $\rho_h(x) = \lambda\int_0^{x_o} xf(x)dx = 1$.

In Figure 10 we evaluate our analytic expressions under the specific distribution of request sizes from our trace workload. Below we explain this figure.

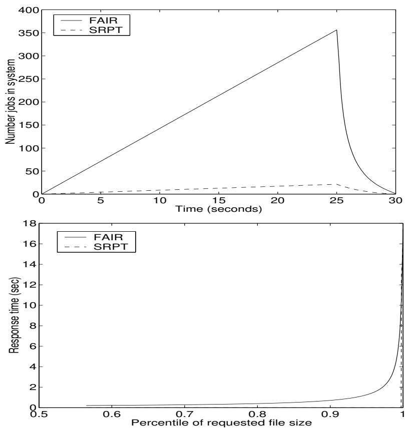


Figure 10: *Analytically-derived: Number of requests as a function of time under FAIR and SRPT. Parameters used:* $\rho_h = 1.2, \rho_l = 0, t_h = 25ms, t_l = 25ms$*, request size distribution is close fit to request distribution in trace.*

Figure 10(left) corresponds to one cycle in Figure 6 (implementation). From Figure 10 we see that the number of requests in the M/G/1/FAIR queue is significantly higher than in the M/G/1/SRPT queue, which is the same trend shown in Figure 6. The exact numbers are lower in the analysis (Figure 10) as compared with the implementation (Figure 6). This is to be expected because the analysis assumes smaller service times for a request (since it does not factor in TCP delays and processing time of the request outside of transmission time), as explained above.

Figure 10(right) should correspond to Figure 7(right). In both the analysis and the implementation, most requests have far lower response times under SRPT as compared with FAIR scheduling. In both the analysis and in the implementation, it turns out that the very largest requests see approximately the same mean response time under SRPT as compared with FAIR. In the analysis we considered requests up through the 99.999 percentile. Even these requests saw slightly lower mean response time under SRPT as compared with FAIR.

10 Appendix B: Other Traces

In this appendix we consider one more trace and run all experiments on the new trace. We find that the results are virtually identical to those shown in the body of the paper.

The log used here was collected from a NASA web server and is also available through the Internet Traffic Archive [27]. 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 lies around 1.9 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 11, 12 and 13. They are extremely similar to the corresponding Figures 6, 8 and 9 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 however are similar. The increase in response times under the NASA log may be attributed to the higher mean file size.

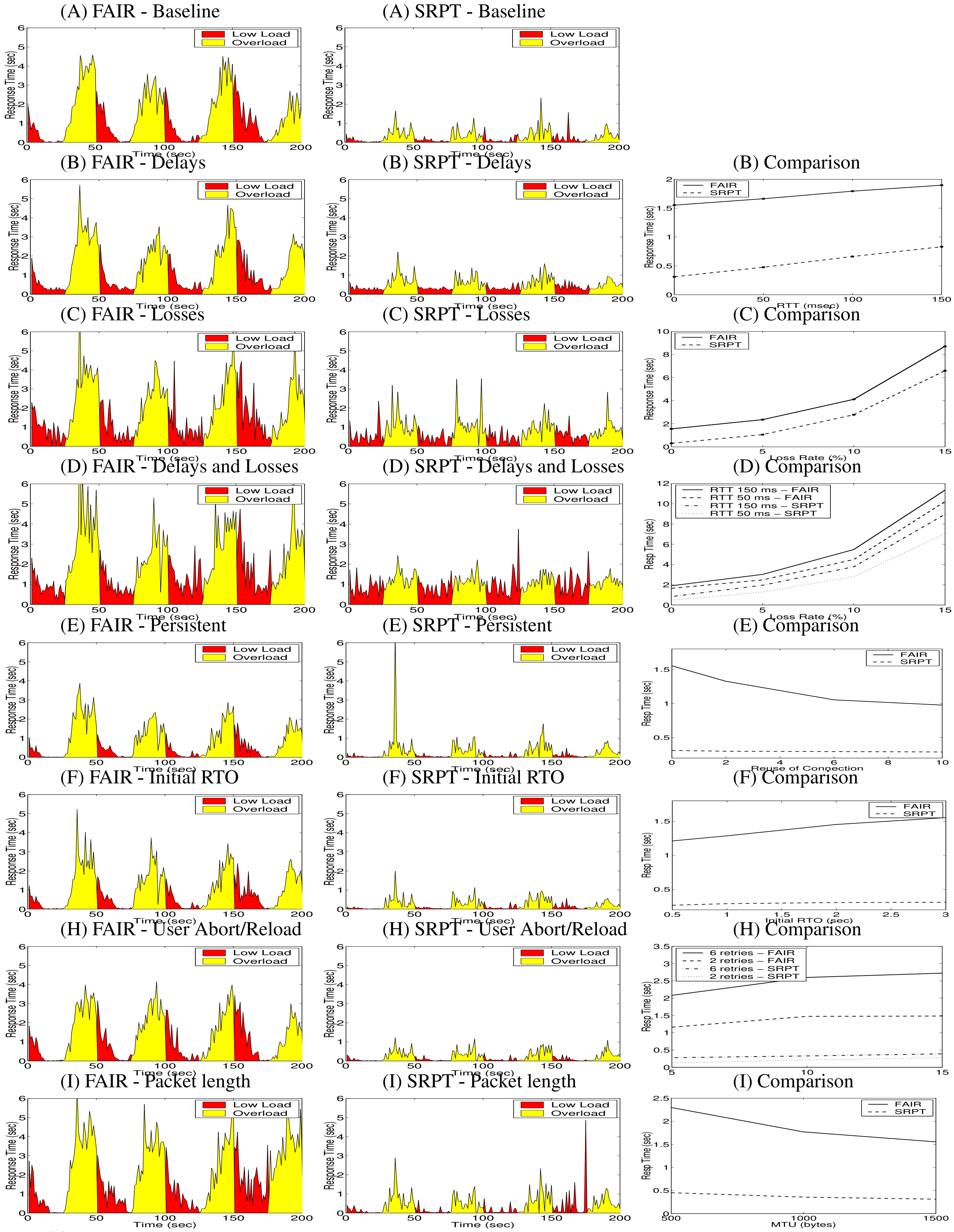
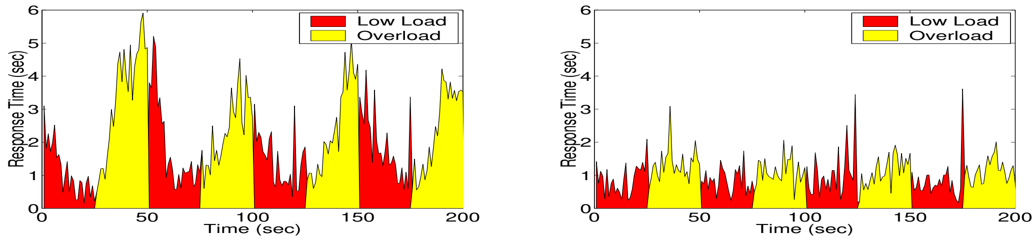
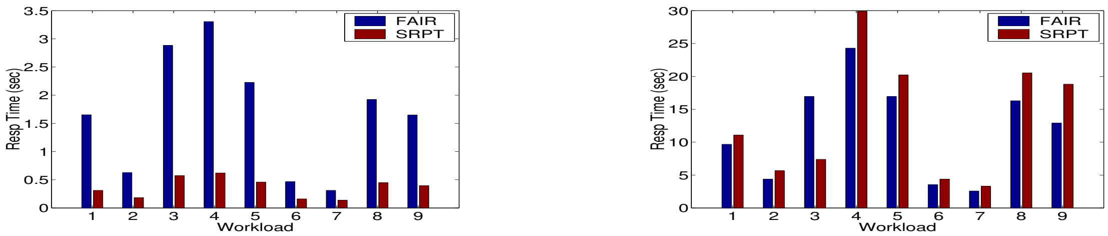


Figure 11: *Results under NASA trace log. Each row in the figure compares SRPT and FAIR under workload W1 for one particular setup from Table 2. The left and middle column in the figure show the response times over time for the specific values given in Table 2, column 3. The right column in the figure compares the performance of SRPT and FAIR for the range of values given in the column 4 of Table 2.*



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

Figure 12: *Comparison of FAIR (left) and SRPT (right) for the realistic setup for workload W1, under NASA trace log.*



Mean Response Time - Baseline Mean Response Time of biggest 1% requests - Baseline

Figure 13: *Comparison of FAIR and SRPT in the baseline case for the workloads in Table 1, under the NASA trace log. The mean response times over all requests are shown left, and the mean response times of only the biggest 1% of all requests are shown right.*