Connection Scheduling in Web Servers

Mark E. Crovella* Robert Frangioso*
Department of Computer Science
Boston University
Boston, MA 02215
{crovella,rfrangio}@bu.edu

Mor Harchol-Balter†
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
harchol@cs.cmu.edu

Abstract

Under high loads, a Web server may be servicing many hundreds of connections concurrently. In traditional Web servers, the question of the order in which concurrent connections are serviced has been left to the operating system. In this paper we ask whether servers might provide better service by using non-traditional service ordering. In particular, for the case when a Web server is serving static files, we examine the costs and benefits of a policy that gives preferential service to short connections. We start by assessing the scheduling behavior of a commonly used server (Apache running on Linux) with respect to connection size and show that it does not appear to provide preferential service to short connections. We then examine the potential performance improvements of a policy that does favor short connections (shortest-connection-first). We show that mean response time can be improved by factors of four or five under shortest-connection-first, as compared to an (Apache-like) size-independent policy. Finally we assess the costs of shortest-connection-first scheduling in terms of unfairness (*i.e.,* the degree to which long connections suffer). We show that under shortest-connection-first scheduling, long connections pay very little penalty. This surprising result can be understood as a consequence of heavy-tailed Web server workloads, in which most connections are small, but most server load is due to the few large connections. We support this explanation using analysis.

1 Introduction

As the demand placed on a Web server grows, the number of concurrent connections it must handle increases. It is not uncommon for a Web server under high loads to be servicing many hundreds of connections at any point in time. This situation raises the question: when multiple outstanding connections require service, in what order should service be provided?

By service, we mean the use of some system device (processor, disk subsystem, or network interface) that allows the server to make progress in delivering bytes to the client associated with the connection. Thus, the question of service order applies collectively to the order in which connections are allowed to use the CPU, the disk(s), and the network interface.

In most Web servers, the question of the order in which concurrent connections should be serviced has typically been left to a general-purpose operating system. The OS scheduler orders access to the CPU, and the disk and network subsystems order service requests for disk and network I/O, respectively. The policies used in these systems typically emphasize fairness (as provided by, *e.g.,* approximately-FIFO service of I/O requests) and favorable treatment of interactive jobs (as provided by feedback-based CPU scheduling).

In this paper, we examine whether Web servers might provide better service by using non-traditional service ordering for connections. In particular, we are concerned with Web servers that serve static files. In this case, the service demand of the connection can be accurately estimated at the outset (*i.e.,* once the HTTP GET has been received) since the size of the file to be transferred is then known; we call this the "size" of the connection. The question then becomes: can servers use the knowledge of connection size to improve mean response time?

Traditional scheduling theory for simple, single-device systems shows that if task sizes are known, policies that favor short tasks provide better mean response time than policies that do not make use of task size. In a single-device system, if run-

*Supported in part by NSF Grant CCR-9706685.
†Supported by the NSF Postdoctoral Fellowship in the Mathematical Sciences.

ning jobs can be pre-empted, then the optimal work-conserving policy with respect to mean response time is *shortest remaining processing time first* (SRPT). Since a Web server is not a single-device system, we cannot use SRPT directly. However we can employ service ordering within the system that attempts to approximate the effects of SRPT.

The price to be paid for reducing mean response time is that we reduce the fairness of the system. When short connections are given favorable treatment, long connections will suffer. Care must be taken to ensure that the resulting unfairness does not outweigh the performance gains obtained.

Our goal in this paper is to explore the costs and benefits of service policies that favor short connections in a Web server. We call this the *connection scheduling* problem. The questions we address are:

1. How does a traditional Web server (Apache running on Linux) treat connections with respect to their size? Does it favor short connections?

2. What are the potential performance improvements of favoring short connections in a Web server, as compared to the traditional service order?

3. Does favoring short connections in a web server lead to unacceptable unfairness?

We are interested in the answers to these questions in the context of a traditionally structured operating system (like Linux). Thus, to answer these questions we have implemented a Web server, running on Linux, that allows us to experiment with connection scheduling policies. For each of the devices in the system (CPU, disk, and network interface) the server allows us to influence the order in which connections are serviced. Since all of our scheduling is done at the application level, our server does not allow us precise control of all of the components of connection service, particularly those that occur in kernel mode; this is a general drawback of traditional operating systems structure whose implications we discuss in detail below. However our server does provide sufficient control to allow us to explore two general policies: 1) *size-independent* scheduling, in which each device services I/O requests in roughly the same order in which they arrive; and 2) *shortest-connection-first* scheduling, in which each device provides service only to the shortest connections at any point in time. We use the SURGE workload generator [5] to create Web requests; for our purposes, the important property of SURGE is that it accurately mimics the size distribution of requests frequently seen by Web servers.

Using this apparatus, we develop answers to the three questions above. The answer to our first question is that Apache does *not* appear to favor short connections. We show that compared to our server's size-independent policy, Apache's treatment of different connection sizes is approximately the same (and even can be more favorable to long connections—a trend opposite to that of shortest-connection-first).

This result motivates our second question: How much performance improvement is possible under a shortest-connection-first scheduling policy, as compared to size-independent (*i.e.,* Apache-like) scheduling? We show that for our server, adopting shortest-connection-first can improve mean response time by a factor of 4 to 5 under moderate loads.

Finally our most surprising result is that shortest-connection-first scheduling does *not* significantly penalize long connections. In fact, even very long connections can experience improved response times under shortest-connection-first scheduling when compared to size-independent scheduling. To explore and explain this somewhat counterintuitive result we turn to analysis. We use known analytic results for the behavior of simple queues under SRPT scheduling, and compare these to size-independent scheduling. We show that the explanation for the mild impact of SRPT scheduling on long connections lies in the size distribution of Web files—in particular, the fact that Web file sizes show a *heavy tailed* distribution (one whose tail declines like a power-law). This result means that Web workloads are particularly well-suited to shortest-connection-first scheduling.

2 Background and Related Work

The work reported in this paper touches on a number of related areas in server design, and in the theory and practice of scheduling in operating systems.

Traditional operating system schedulers use heuristic policies to improve the performance of short tasks given that task sizes are not known in advance. However, it is well understood that in the case where the task sizes *are* known, the work-conserving scheduling strategy that minimizes mean response time is shortest-remaining-processing-time first (SRPT). In addition to SRPT, there are many algorithms in the literature which are designed for the case where the task size is known. Good overviews of the single-node scheduling problem and its solution are given in [7], [14], and [17].

In our work we focus on servers that serve static content, *i.e.,* files whose size can be determined in advance. Web servers can serve dynamic content as well; in this case our methods are less directly applicable. However, recent measurements have suggested that most servers serve mainly static content, and that dynamic content is served mainly from a relatively small fraction of the servers in the Web [15].

Despite the fact that the file sizes are typically available to the Web server, very little work has considered size-based scheduling in the Web. One paper that does discuss size-based scheduling in the Web is that of Bender, Chakrabarti, and Muthukrishnan [6]. This paper raises an important point: in choosing a scheduling policy it is important to consider not only the scheduling policy's performance, but also whether the policy is fair, *i.e.* whether some tasks have particularly high slowdowns (where slowdown is response time over service time). That paper considers the metric *max slowdown* (the maximum slowdown over all tasks) as a measure of unfairness. The paper proposes a new algorithm, *Dynamic Earliest Deadline First (DEDF)*, designed to perform well on both the mean slowdown and max slowdown metrics. The DEDF algorithm is a theoretical algorithm which cannot be run within any reasonable amount of time (it requires looking at all previous arrivals), however it has significance as the first algorithm designed to simultaneously minimize max slowdown and mean slowdown. That work does consider a few heuristics based on DEDF that are implementable; however, simulation results evaluating those more practical algorithms at high load indicate their performance to be about the same as SRPT with respect to max slowdown and significantly worse than SRPT with respect to mean slowdown.

At the end of our paper (Section 6) we turn to analysis for insight into the behavior of the shortest-connection-first scheduling policy. In that section we examine a single queue under SRPT scheduling, which was analyzed by Schrage and Miller [18].

In addition to scheduling theory, our work also touches on issues of OS architecture. In particular, the work we describe in this paper helps to expose deficiencies in traditional operating system structure that prevent precise implementation of service policies like shortest-connection-first. Shortest-connection-first scheduling requires that resource allocation decisions be based on the connection requiring service. This presents two problems: first, kernel-space resource allocation is not under the control of the application; and second, resource allocation is difficult to perform on a per-connection basis. These two problems have been have been noted as well in other work [1, 3].

3 A Server Architecture for Scheduling Experiments

In this section we present the architecture of the web server we designed and developed for use in our experiments. Our primary goal in designing this server was to provide the ability to study policies for scheduling system resources. Two additional but less important goals were simplicity of design and high performance.

3.1 Scheduling Mechanisms

Web servers like Apache [11] follow the traditional architectural model for Internet service daemons, in which separate connections are served by separate Unix processes. This model is insufficient for our needs because none of its scheduling decisions are under application control. Instead we need to expose scheduling decisions to the application as much as possible.

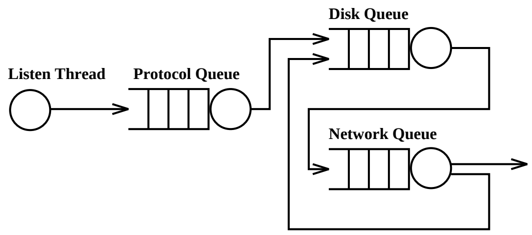

Figure 1: Organization of the Experimental Server

A typical connection needs three types of service after connection acceptance: 1) protocol processing, 2) disk service, and 3) network service. In order to expose the scheduling decisions associated with each type of service, we organize the server application as a set of three queues. This architecture is shown in Figure 1. The entities that are held in (and move between) queues correspond to individual connections. Next we describe how, by varying the service order of each queue, the application can influence resource scheduling decisions on a per-connection basis.

Each queue in Figure 1 has an associated pool of threads. In addition there is a single listen thread. The role of the listen thread is to block on the `accept()` call, waiting for new connections. When a new connection arrives, it creates a *connection descriptor*. The connection descriptor encapsulates the necessary state the connection will need (two file descriptors, a memory buffer, and progress indicators). It then places the connection descriptor into the protocol queue and resumes listening for new connections.

Protocol threads handle all aspects of HTTP. When a protocol thread is done, the server knows what file is being requested and is ready to start sending data. In addition the thread has called `stat()` on the file to determine the file's size. The protocol thread then enqueues the connection descriptor into the disk queue.

The role of the disk thread is to dequeue a connection descriptor, and based on the file associated with the connection, `read()` a block of file data from the filesystem into the connection descriptor's buffer. Currently our server reads blocks of up to 32KB at a time. The `read()` call is blocking; when it returns, the thread enqueues the descriptor into the network queue.

The network thread also starts by dequeueing a connection descriptor; it then calls `write()` on the associated socket to transfer the contents of the connection's buffer to the kernel's socket buffer. The `write()` call is blocking. When it returns, if the all the bytes in the file have been transferred, the network thread will close the connection; otherwise it will place the descriptor back into the disk queue. Thus each connection will move between the network and disk queues until the connection has been entirely serviced.

An important advantage of this architecture is that we can observe which subsystem (protocol, disk, or network) is the bottleneck by inspecting the lengths of the associated queues. For the workloads we used (described in Section 4.1) we found that the bottleneck was the network queue; queue lengths at the protocol and disk queues were always close to zero.

This scheme gives us a flexible environment to influence connection scheduling by varying the service order at each queue. In this study we focus on two scheduling policies: *size-independent* and *shortest-connection-first.* In our size-independent policy, each thread simply dequeues items from its associated queue in FIFO order. The implication of this policy is that each connection is given a fair share of `read()` and `write()` calls, and that any fixed set of connections is conceptually served in approximately round-robin order.

Under shortest-connection-first scheduling, each (disk or network) thread dequeues the connection that has the least number of bytes remaining to be served. This appears to be a good indicator of the remaining amount of work needed to service the connection. More precise policies are possible, which are not considered in this paper.

Finally, we note that the listen and protocol threads run at a higher priority than disk and network threads, and that the protocol queue is always served in FIFO order (since connection size is not yet known).

3.2 Performance

As stated at the outset, high performance is only a secondary goal of our archtectural design. However our architecture is consistent with recent trends in high performance servers and (as shown in Section 5) yields performance that is competitive with a more sophisticated server (Apache).

Considerable attention has been directed to improving the architecture of high performance web servers. As mentioned above, servers like Apache follow the model in which separate connections are served by separate Unix processes. More recent servers have moved away from process-per-connection model, toward lower-overhead strategies. A number of Web server architectures based on a single or fixed set of processes have been built [13, 16]. Removing the overhead of process creation, context switching, and inter-process-communication to synchronize and dispatch work allows the server to use system resources more efficiently. In addition, in single-process servers memory consumption is reduced by not using a running process for each concurrent connection receiving service, which allows such servers to make more effective use of memory to cache data. The drawback is that single process web servers are typically more complicated and must rely on multi-threading or non-blocking I/O schemes to achieve high throughput.

Our server obtains the performance benefits of using a single process, with a fixed number of threads (*i.e.,* it does not use a thread per connection). However we have not adopted all of the performance enhancements of aggressively optimized servers like Flash [16] because of our desire to keep the server simple for flexibility in experimenting with scheduling policies. In particular we use blocking threads for writing, which is not strictly necessary given an nonblocking I/O interface. However we note that the policies we explore in our server appear to be easily implementable in servers like Flash.

3.3 Limitations

The principal limitation of our approach is that we do not have control over the order of events inside the operating system. As a specific example, consider the operation of the network subsystem. Our server makes `write()` calls which populate socket buffers in the network subsytem in a particular order. However, these buffers are not necessarily drained (*i.e.,* written to the network) in the order in which our application has filled them.

Two factors prevent precise control over the order in which data is written to the network. First, each buffer is part of a flow-controlled TCP connection with a client. If the client is slow with respect to the server, the client's behavior can influence the order in which buffers are drained. For this reason in our experiments we use a multiple high-performance clients and we ensure that the clients are not heavily loaded on average. Thus in our case client interaction is not a significant impediment to scheduling precision.

The second, more critical problem is that in traditional Unix network stack implementations, processing for all connections is handled in an aggregate manner. That is, outgoing packets are placed on the wire in response to the arrival of acknowledgements. This means that if many connections have data ready to send, and if the client and network are not the bottleneck, then data will be sent from the set of connections in order that acknowledgements arrive, which is not under application control. The implication is that if the network subsystem has a large number of connections that have data ready to send, then the order in which the application has written to socket buffers will have less effect on the scheduling of connections.

The problem has been recognized and addressed in previous work. In particular, Lazy Receiver Processing [10] can isolate each connection's path through the network stack. This allows scheduling decisions to be made on a per-connection basis at the level of the network interface.

Our goal was to demonstrate the improvements possible without operating system modification. As a result, to obtain control over I/O scheduling we limit the concurrency in the I/O systems. For example, by limiting concurrency in the network subsystem, we limit the number of connections that have data ready to send at any point in time, thus narrowing the set of connections that can transmit packets. Because our disk and network threads use blocking I/O, we need multiple threads if we want to have concurrent outstanding I/O requests. This means that it is straightforward to control the amount of concurrency we allow in the kernel subsystems, by varying the number of threads in each of the pools.

At one extreme, if we allow only one thread per pool, then we have fairly strict control over the order of events inside the kernel. At any point in time the kernel can only have one I/O request of each type pending, so there are no scheduling decisions available to it. Unfortunately this approach sacrifices throughput; both the disk and network subsystems make use of concurrent requests to overlap processing with I/O. At the other extreme, if we provide a large number of threads to each pool, we can obtain high throughput; however then we lose all control over scheduling.

In order to explore the utility of shortest-connection-first scheduling, we have adopted an intermediate approach. Rather than running our server at its absolute maximum throughput (as measured in bytes per unit time), we limit its throughput somewhat in order to obtain control over I/O scheduling. Note however that our server's throughput (in bytes per second) is still greater than that of Apache under the same load level.[2] The performance implications of this approach are presented in Section 5.4. It is important to note that this is only necessary because of the limitations of the traditionally structured OS on which we run our experiments, and this restriction could be dropped given a different OS structure. Our approach thus allows us enough influence over kernel scheduling to demonstrate the costs and benefits of the shortest-connection-first policy.

4 Experimental Setup

4.1 File Size Distribution

An important aspect of our work is that we have focused on careful modeling of the file size distribution typically seen on Web servers. As shown in Section 6, properties of the file size distribution are directly related to some of our results.

Our starting point is the observation that file sizes on Web servers typically follow a *heavy-tailed* distribution. This property is surprisingly ubiquitous in the Web; it has been noted in the sizes of files requested by clients, the lengths of network connections, and files stored on servers [2, 8, 9]. By heavy tails we mean that the tail of the empirical distribution function declines like a power law with exponent less than 2. That is, if a random variable X follows a heavy-tailed distribution, then

$$P[X > x] \sim x^{-\alpha}, \quad 0 < \alpha < 2$$

where $f(x) \sim a(x)$ means that $\lim_{x\to\infty} f(x)/a(x) = c$ for some positive constant c.

[2] Apache was configured for high performance as described in Section 4.2.

	Body
Distribution	Lognormal
PMF	$\frac{1}{x\sigma\sqrt{2\pi}}e^{-(\ln x-\mu)^2/2\sigma^2}$
Range	$0 \leq x < 9020$
Parameters	$\mu = 7.630$; $\sigma = 1.001$
	Tail
Distribution	Bounded Pareto
PMF	$\frac{\alpha k^\alpha}{1-(k/p)^\alpha}x^{-\alpha-1}$
Range	$9020 \leq x \leq 10^{10}$
Parameters	$k = 631.44$; $\alpha = 1.0$; $p = 10^{10}$

Table 1: Empirical Task Size Model. PMF is the probability mass function, $f(x)$, where $\int_a^b f(x)dx$ represents the probability that the random variable takes on values between a and b.

Random variables that follow heavy tailed distributions typically show extremely high variability in size. This is exhibited as many small observations mixed with a small number of very large observations. The implication for Web files is that a tiny number of the very largest files make up most of the load on a Web server. We refer to this as the *heavy-tailed property* of Web task sizes; it is central to the discussion in this paper and will come up again in Section 6.

Although Web files typically show heavy tails, the body of the distribution is usually best described using another distribution. Recent work has found that a hybrid distribution, consisting of a body following a lognormal distribution and a tail that declines via a power-law, seems to fit well some Web file size measurements [4, 5]. Our results use such a model for task sizes, which we call the *empirical* model; parameters of the empirical model are shown in Table 1. In this empirical file size model, most files are small—less than 5000 bytes. However, the distribution has a very heavy tail, as determined by the low value of α in the Bounded Pareto distribution, and evidenced by the fact that the mean of this distribution is 11108 — much larger than the typical file size.

4.2 Experimental Environment

To generate HTTP requests that follow the size distribution described above, we use the SURGE workload generator [5]. In addition to HTTP request sizes, SURGE's stream of HTTP requests also adheres to empirically derived models for the sizes of files stored on the server; for the relative popularity of files on the server; for the temporal locality present in the request stream; and for the timing of request arrivals at the server.

SURGE makes requests using synthetic clients, each of which operates in a loop, alternating between requesting a file and lying idle. Each synthetic client is called a *User Equivalent* (UE). The load that SURGE generates is varied by varying the number of UEs. In our tests we varied the number of UEs from 400 to 2000. Validation studies of SURGE are presented in [5]; that paper shows that the request stream created by SURGE conforms closely to measured workloads and is much burstier, and hence more realistic, than that created by SPECWeb96 (a commonly used Web benchmarking tool).

All measurements of Apache's performance presented in this paper were generated using version 1.2.5. We configured Apache for high performance as recommended on Apache's performance tuning Web page.[3] In particular, `MaxRequestsPerChild` was set to 0, meaning that there is no fixed limit to the number of connections that can be served by any of Apache's helper processes. This setting improves Apache's performance considerably as compared to the default.

In addition to the data reported in this paper, we also ran many experiments using version 1.3.4 of Apache. Our experiments indicated that this version had a performance anomaly under low load that we did not isolate, so we do not present those results. However, our experiments indicated that although version 1.3.4 was somewhat faster for short connections, its overall performance was not different enough to affect the conclusions in this paper.

All our tests were conducted using two client machines (evenly splitting the SURGE UEs) and one server in a 100 Mbit switched Ethernet environment. After our experiments were concluded, we found that one client's port was malfunctioning and delivering only 10 Mbit/sec; so half of the load was generated over a path that was effectively only 10Mbit/sec, while the other half of the load arrived over a 100Mbit/sec path. All machines were Dell Dimensions equipped with Pentium II 233 processors, 128 MB of RAM, and SCSI disks. Each of these machines was running Linux 2.0.36. We configured SURGE to use a file set of 2000 distinct files varying in size from 186 bytes to 121 MB. Our measurements pertaining to response time, byte throughput, and HTTP GETs per second were extracted from client side logs generated by SURGE.

All experiments were run for ten minutes. This time was chosen to be sufficiently long to provide

[3] `http://www.apache.org/docs/misc/perf-tuning.html`.

confidence that the measurements were not strongly influenced by transients. For a 1000 UE experiment, this meant that typically more than 2.5GB of data was transferred and more than 250,000 connections took place.

5 Results

5.1 Characterizing Apache's Performance as a Function of Task Size

In this section we characterize Apache's performance as a function of task size under 3 different load levels: 1000 UEs, 1400 UEs, and 1800 UEs. These loads correspond to lightly loaded, moderately loaded, and overloaded conditions for the server. Thus they span the range of important loads to study.

In order to study how different servers treat connections with respect to size, we bin HTTP transactions according to size, and plot the mean response time over all transactions in the bin, as a function of mean file size of transactions in the bin. We plot the resulting data on log-log axes, in order to simultaneously examine both very small and very large connection sizes. Bin sizes grow exponentially, leading to equal spacing on logarithmic axes.

Figure 2 shows the resulting plots of mean response time as a function of file size under Apache and under our Web server with size-independent scheduling for the three different load levels. The plots generally show that reponse time of small files (less than about 10KB) is nearly independent of file size. For this range of files, response time is dominated by connection setup cost. For large files (larger than about 10KB) reponse time increases as a function of file size in an approximately linear manner.

Across all loads, two trends are evident from the figure. First, for small files, Apache tends to provide the same response time or worse response time than does our size-independent server. Second, for large files, Apache tends to provide the same reponse time or better than does our size-independent server.

These two trends indicate that with respect to size, Apache treats connections in a manner that is either approximately the same as our size-independent policy, or else is more favorable to long connections.[4] That is, Apache is, if anything, *punishing* short connections with respect to our size-independent server.

[4] As discussed in Section 4.2 these measurements use Apache version 1.2.5. We found in other experiments with Apache 1.3.4 (not shown here) that the newer version of Apache in fact shows performance that is even closer to that of our server with size-independent scheduling.

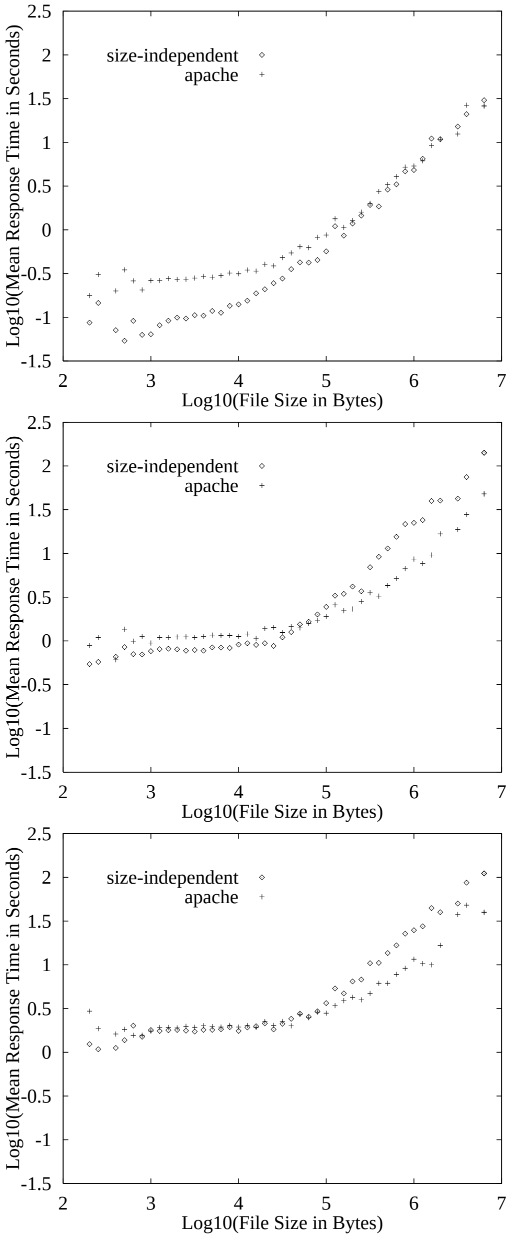

Figure 2: Mean transfer time as a function of file size under the Apache server and our server with size-independent scheduling. Both axes use a log scale. Top to bottom: 1000, 1400, and 1800 UEs.

5.2 Performance Improvements Possible with Shortest-Connection-First Scheduling

Given that Apache does not appear to treat connections in a manner that is favorable to short connections, our next question is whether a policy that does favor short connections leads to performance improvement, and if so, how much improvement is possible. Thus, in this section we compare the

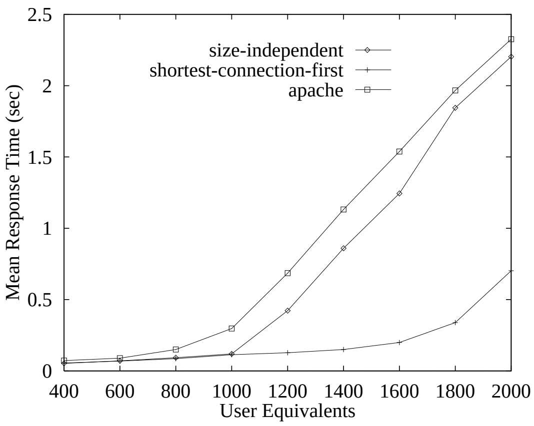

Figure 3: Mean response time as a function of load for our Web server with shortest-connection-first scheduling and size-independent scheduling, and for Apache.

mean response time of our Web server with shortest-connection-first scheduling versus size-independent scheduling.

Figure 3 shows mean response time as a function of the number of user equivalents for our server with shortest-connection-first scheduling compared with size-independent scheduling. The figure shows that at low loads (less than 1000 UEs) there is no difference between the two scheduling policies. Thus 1000 UEs represents the point where the network queue in our server first starts to grow, making the order in which it services write requests important. As the load increases beyond 1000 UEs, the difference in performance between shortest-connection-first scheduling and size-independent scheduling becomes stark. For example under 1400 UEs, the shortest-connection-first scheduling policy improves mean response time by a factor of 4 to 5 over the size-independent scheduling policy.

Also plotted for reference in Figure 3 is the mean response time of Apache under the same conditions. As can be seen, Apache's performance is very similar to that of our server with size-independent scheduling. This is consistent with the conclusions from the previous subsection.

It is important to note that these performance figures may only be lower bounds on the improvement possible by using shortest-connection-first scheduling, due to our constraint of working within a traditionally structured operating system.

5.3 How Much Do Long Connections Suffer?

In the previous section we saw that large improvements in mean transfer time were possible by running our Web server under shortest-connection-first scheduling as opposed to size-independent scheduling. The question now is: does this performance improvement come at a significant cost to large jobs? Specifically, we ask whether large jobs fare worse under shortest-connection-first scheduling than they do under size-independent scheduling.

To answer this question we examine the performance of both shortest-connection-first scheduling and size-independent scheduling as a function of task size. The results are shown in Figure 4, again for a range of system loads (UEs).

The figure shows that in the case of 1000 UEs, shortest-connection-first scheduling is identical in performance to size-independent scheduling across all file sizes. Thus since there is no buildup at the network queue, there is also no performance improvement from shortest-connection-first scheduling in the case of 1000 UEs. However, the figure shows that in the case of 1400 UEs, shortest-connection-first results in much better performance for small jobs (as compared with size-independent scheduling), and yet the large jobs still do not fare worse under shortest-connection-first scheduling than they do under size-independent scheduling. Thus the overall performance improvement does *not* come at a cost in terms of large jobs. When we increase the load to 1800 UEs, however, the large jobs do begin to suffer under shortest-connection-first scheduling as compared with size-independent scheduling. In fact over all our experiments, we find that in the range from about 1200 UEs to 1600 UEs, shortest-connection-first allows short connections to experience considerable improvement in response time without significantly penalizing long connections. This seemingly counter-intuitive result is explained, with analytical justification, in Section 6.

5.4 Varying The Write Concurrency

As discussed in Section 3.3, in order to gain control over the order in which data is sent over the network, we need to restrict our server's throughput (in bytes per second). In this section we quantify this effect.

In all our previous plots we have used a thread pool of 35 threads to service the network queue. Figure 5 shows the effect of varying the number of network threads on mean response time and on documents (HTTP GETs) served per second, at a load of 1400 UEs.

Both plots in the figure make the point that as the number of write threads increases, the difference in performance between shortest-connection-first and size-independent scheduling decreases, un-

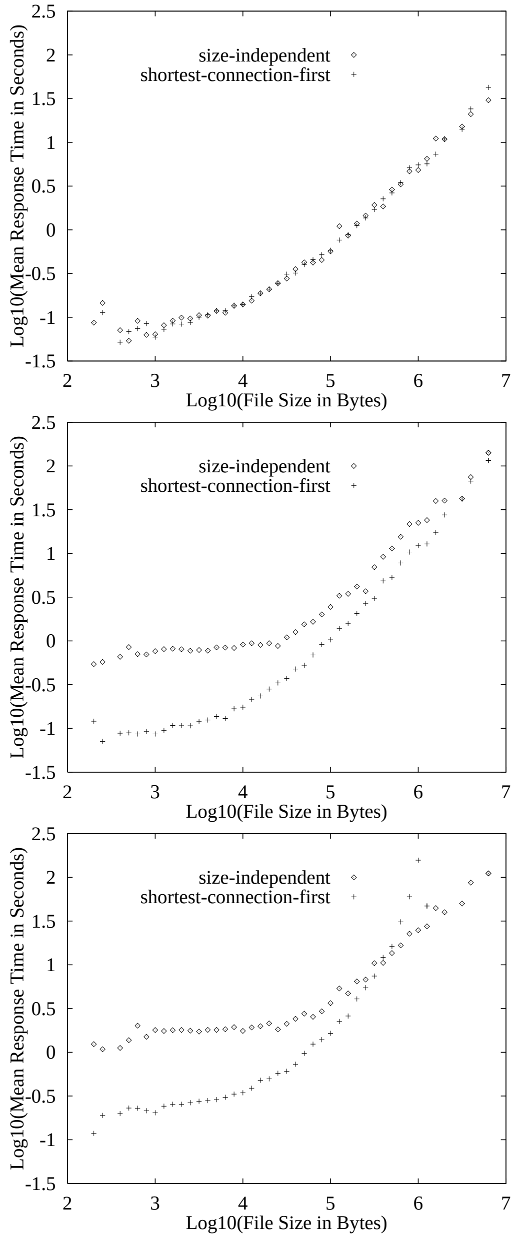

Figure 4: Response time as a function of task size for shortest-connection-first scheduling versus size-independent scheduling. Top to bottom: 1000, 1400, and 1800 UEs.

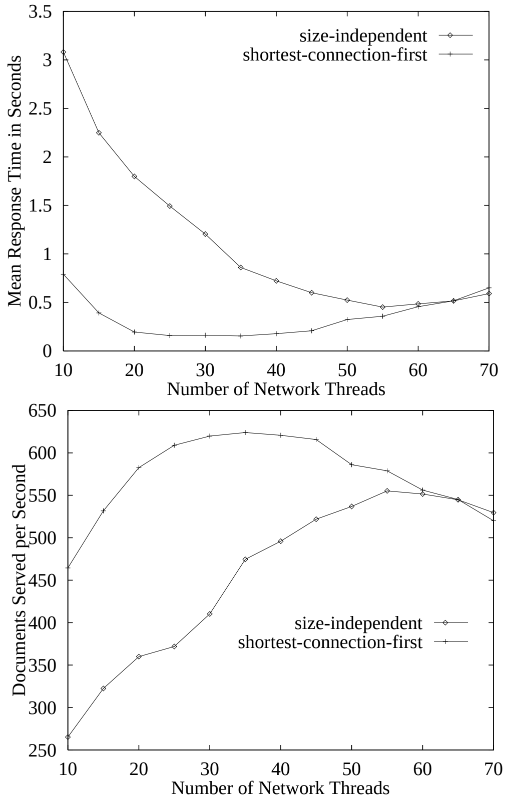

Figure 5: The effect of varying the number of network threads on (upper) mean response time and (lower) HTTP GETs/second for the two scheduling policies.

til at about 60 threads, the choice of scheduling policy has no effect. At the point of 60 threads, there is no buildup in the network queue and all scheduling is determined by kernel-level events.

These plots also show that as the number of threads used declines from 35 threads, the performance difference between shortest-connection-first and traditional scheduling becomes even greater. This suggests that the advantage of shortest-connection-first scheduling may be even more dramatic in a system where there is greater control over kernel-level scheduling, since as the number of threads declines in our system, the degree of control over kernel scheduling increases.

Figure 6 shows the effect on byte throughput of varying the number of network threads. In this figure we have plotted the total number of bytes in files succesfully transferred during each of our 10 minute experiments. It shows that for our server, throughput increases roughly linearly with additional network threads, regardless of the policy used. In all cases, shortest-connection-first has slightly higher throughput than our size-independent policy; this is because the server is serving fewer concurrent connections (on average) and so can provide slightly higher aggregate performance. The primary point to note is when configured with 35 network threads, our server is not performing at peak throughput; this is the price paid for control over network scheduling. As stated earlier, this price is exacted because the kernel does not support per-connection scheduling within the protocol stack.

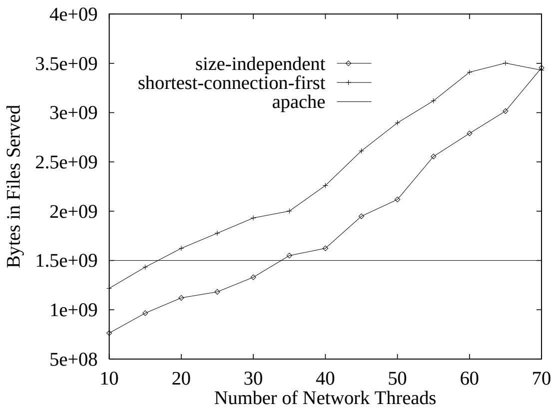

Figure 6: The effect of varying the number of network threads on the byte throughput for the two scheduling policies.

Also plotted for reference in Figure 6 is the corresponding byte throughput of Apache at 1400 UEs. The comparison illustrates that, for 35 network threads, our server is achieving higher throughput than Apache. Thus, although using 35 threads limits our server from its maximum possible performance, it is a level that still outperforms Apache.

6 Analysis: Why Don't Large Jobs Suffer?

In this section we help explain why long connections aren't severely penalized in our experiments using some simple analytic models. These models are only approximations of the complex systems comprising a Web server, but they yield conclusions that are consistent with our experimental results and, more importantly, allow us to explore the reasons behind those experimental results. The results in this section are based on [12]; that paper presents additional background and more results not shown here.

6.1 Assumptions Used in Analysis

In our analysis we examine the $M/G/1$ queue, which is a simple queue fed by a Poisson arrival stream with an arbitrary distribution of service times. The service order is shortest-remaining-processing-time first (SRPT).

Under the SRPT model, only one task at each instant is receiving service, namely, the task with the least processing time remaining. When a new task arrives, if its service demand is less than the remaining demand of the task receiving service, the current task is pre-empted and the new task starts service. We use SRPT as an idealization of shortest-connection-first scheduling because in both cases, tasks with small remaining processing time are always given preference over tasks with longer remaining processing time.

Our analytical results throughout are based on the following equation for the mean response time for a task of size x in an $M/G/1$ queue with load ρ, under SRPT [18]:

$$E\{R_x^{SRPT}\} = \frac{\lambda \int_0^x t^2\,dF(t) + \lambda x^2(1-F(x))}{2\left(1-\lambda\int_0^x t\,dF(t)\right)^2} + \int_0^x \frac{1}{(1-(\lambda\int_0^t z\,dF(z)))}dt$$

where R_x is the response time (departure time minus arrival time) of a job of size x, $F(\cdot)$ is the cumulative distribution function of service time, and λ is the arrival rate.

We adopt the assumption that the amount of work represented by a Web request is proportional to the size of the file requested. Thus, we use as our task size distribution the empirical file size distribution as shown in Table 1 (the same as that generated by SURGE).

6.2 Understanding the Impact on Long Connections

Using our simple models we can shed light on why large tasks do not experience significant penalties under SRPT scheduling; the explanation will apply equally well to long connections in a Web server employing shortest-connection-first.

The key observation lies in the *heavy-tailed* property of the workload being considered. As defined in Section 4.1, this means that a small fraction of the largest tasks makes up most of the arriving work. The Bounded Pareto distribution, which makes up the tail of our empirical distribution, is extremely heavy-tailed. For example, for a Bounded Pareto distribution with $\alpha = 1.1$, the largest 1% of all tasks account for more than half the total service demand arriving at the server. For comparison, for an exponential distribution with the same mean, the largest 1% of all tasks make up only 5% of the total demand. Now consider the effect on a very large task arriving at a server, say a task in the 99th percentile of the job size distribution. Under the Bounded Pareto distribution, this task in the 99th percentile is interrupted by less than 50% of the total work arriving. In comparison, under the exponential distribution, a task in the 99th percentile of the job size distribution is interrupted by about 95% of the total work arriving. Thus, under the

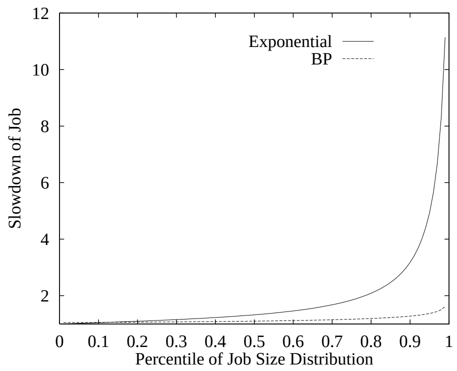

Figure 7: Mean slowdown under SRPT as a function of task size; $\rho = 0.9$.

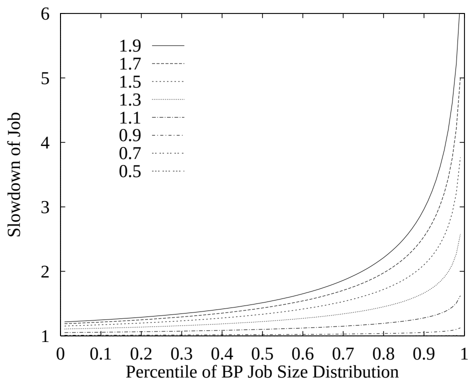

Figure 8: Mean slowdown under SRPT as a function of task size, varying α of task size distribution.

heavy-tailed distribution, a large job suffers *much* less than under a distribution like the exponential. The rest of this section explains and supports this observation.

To evaluate the potential for unfairness to large tasks we plot the mean slowdown of a task of a given size, as a function of the task size. Slowdown is defined as the ratio of a task's response time to its service demand. Task size is plotted in percentiles of the task size distribution, which allows us to assess what fraction of largest tasks will achieve mean slowdown greater than any given value.

Figure 7 shows mean slowdown as a function of task size under the SRPT discipline, for the case of $\rho = 0.9$. The two curves represent the case of an exponential task size distribution, and a Bounded Pareto (BP) task size distribution with $\alpha = 1.1$. The two distributions have the same mean.

Figure 7 shows that under high server load ($\rho = 0.9$), there can be considerable unfairness, but *only for the exponential distribution.* For example, the largest 5% of tasks under the exponential distribution all experience mean slowdowns of 5.6 or more, with a non-negligible fraction of task sizes experiencing mean slowdowns as high as 10 to 11. In contrast, *no* task size in the BP distribution experiences a mean slowdown of greater than 1.6. Thus, when the task size distribution has a light tail (exponential), SRPT can create serious unfairness; however when task size distributions show a heavy tail (BP distribution), SRPT does not lead to significant unfairness.

To illustrate the effect of the heavy-tailed property on the degree of unfairness experienced by large jobs, we plot mean slowdown as a function of task size over a range of BP task size distributions with constant mean (in this case, 3000) and varying α. This plot is shown in Figure 8. The high α cases represent relatively light tails, whereas the low α cases represent relatively heavy tails in the task size distribution.

This figure shows how the degree of unfairness under SRPT increases as the tail weight of the task size distribution decreases. When α is less than about 1.5, there is very little tendency for SRPT to penalize large tasks (curves for $\alpha = 0.5$ and $\alpha = 0.7$ stay so close to 1 as to be invisible on the plot). Only as α gets close to 2.0 (*e.g.,* 1.7 or 1.9) is there any significant fraction of tasks that experience high mean slowdowns.

These figures show that as the heavy-tailed property grows more pronounced, unfairness in the system under SRPT diminishes. Thus the explanation for the surprising resistance of the heavy-tailed task size distributions to unfairness under SRPT is an effect of the heavy-tailed property.

7 Conclusion

This paper has suggested that Web servers serving static files may show significant performance improvements by adopting nontraditional service ordering policies. We have examined the behavior of a popular Web server (Apache running on Linux) and found that, with respect to connection size, it appears to provide service similar to a server that uses size-independent scheduling. Furthermore, we have found that significant improvements in mean response time—on the order of factors of 4 to 5—are achievable by modifying the service order so as to treat short connections preferentially. Finally, we have found that such a service ordering does *not* overly penalize long connections. Using analysis, we have shed light on why this is the case, and concluded that the heavy-tailed properties of Web workloads (*i.e.,* that a small fraction of the longest connections make up a large fraction of the total work) make Web workloads especially amenable to shortest-connection-first scheduling.

Our work has a number of limitations and directions for future work. The architecture of our server does not allow us precise control over the scheduling of kernel-mode operations (such as I/O). This prevents us from determining the exact amount of improvement that is possible under scheduling policies that favor short connections. We plan to implement short-connection favoring strategies over a kernel architecture that is better designed for server support [3, 10] in order to assess their full potential.

There is room for better algorithmic design here, since the policy we have explored does not prevent the starvation of jobs in the case when the server is *permanently* overloaded. One commonly adopted solution to this problem is dynamic priority adjustment, in which a job's priority increases as it ages, allowing large jobs to eventually obtain priorities equivalent to those of small jobs. We plan to explore such improved policies, perhaps following the initial work in [6].

While more work needs to be done, our results suggest that nontraditional scheduling order may be an attractive strategy for Web servers that primarily serve static files. In particular, the fact that Web workloads (file sizes and connection lengths) typically show heavy-tailed distributions means that shortest-connection-first policies can allow Web servers to significantly lower mean response time without severely penalizing long connections.

References

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated quality-of-service in Web hosting services. In *Proceedings of the First Workshop on Internet Server Performance*, June 1998.

[2] Martin F. Arlitt and Carey L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, 1997.

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

[4] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web client access patterns: Characteristics and caching implications. *World Wide Web*, 1999.

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

[6] Michael Bender, Soumen 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.

[7] Richard W. Conway, William L. Maxwell, and Louis W. Miller. *Theory of Scheduling*. Addison-Wesley Publishing Company, 1967.

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

[9] Mark E. Crovella, Murad S. Taqqu, and Azer Bestavros. Heavy-tailed probability distributions in the World Wide Web. In *A Practical Guide To Heavy Tails*, pages 3–26. Chapman & Hall, New York, 1998.

[10] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of OSDI '96*, October 1996.

[11] The Apache Group. Apache web server. `http://www.apache.org`.

[12] M. Harchol-Balter, M. E. Crovella, and S. Park. The case for SRPT scheduling in Web servers. Technical Report MIT-LCS-TR-767, MIT Lab for Computer Science, October 1998.

[13] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth MacKenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles*, October 1997.

[14] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. In *CRC Handbook of Computer Science*. 1997.

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

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

[17] M. Pinedo. *On-line algorithms, Lecture Notes in Computer Science*. Prentice Hall, 1995.

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