

The Origins of Network Server Latency & the Myth of Connection Scheduling^{*}

Yaoping Ruan
yruan@cs.princeton.edu

Vivek S. Pai
vivek@cs.princeton.edu

Department of Computer Science, Princeton University, Princeton, NJ 08544

ABSTRACT

We investigate the origins of server-induced latency to understand how to improve latency optimization techniques. Using the Flash Web server [4], we analyze latency behavior under various loads. Despite latency profiles that suggest standard queuing delays, we find that most latency actually originates from negative interactions between the application and the locking and blocking mechanisms in the kernel. Modifying the server and kernel to avoid these problems yields both qualitative and quantitative changes in the latency profiles – latency drops by more than an order of magnitude, and the effective service discipline also improves.

We find our modifications also mitigate service burstiness in the application, reducing the event queue lengths dramatically and eliminating any benefit from application-level connection scheduling. We identify one remaining source of unfairness, related to competition in the networking stack. We show that adjusting the TCP congestion window size addresses this problem, reducing latency by an additional factor of three.

Categories and Subject Descriptors: C.4 PERFORMANCE OF SYSTEMS:Performance attributes

General Terms:Measurement, Performance.

Keywords: Network Server, Latency, Connection Scheduling.

1. INTRODUCTION

Improvements in server-side connectivity, the increasing broadband penetration rate, and the popularity of Web proxy cache servers are reducing the delays associated with wide-area networks. As a result, server-induced latency can become a noticeable fraction of the end user's response latency. Some recent work has begun to address these issues with optimization approaches mostly focusing on connection scheduling [1, 2, 3, 6]. These approaches generally make the assumption that queuing delays are inherent to the system, and that existing operating systems handle requests unfairly or inefficiently. Unfortunately, the existing research does not address *why* these delays exist, complicating attempts to systematically address their origins.

We show that while server latency profiles resemble standard queuing delays, most server-induced latency actually stems from locking and blocking issues in the kernel. These problems cause head-of-line blocking, which hinders the existing fairness mechanisms in the scheduler and networking stack. We fix these problems

^{*}This work has been partially supported by an NSF CAREER award

and demonstrate a new server with much better latency profiles, and more than an order of magnitude reduction in both median and mean latencies. We further discover that eliminating such blocking also reduces the burstiness of event delivery. With a smaller number of ready events at any time, the opportunities for connection scheduling disappear, suggesting that connection scheduling is unnecessary and only a result of other implementation artifacts.

Finally, we address the remaining issues of latency and fairness by examining the networking stack, and demonstrate a new approach to reduce latency that adaptively adjusts the TCP congestion window. We show that it achieves latency benefit effectively but with much less effort than previous approaches. The results from our latency studies demonstrate that when the coarse-grained blocking is avoided, the existing scheduling inside the networking stack adequately performs fine-grained control.

2. THE ORIGIN OF EXCESS LATENCY

Using a popular static content workload with a heavy-tailed distribution, we measure the latency behavior of the Flash server at various load levels¹. These results, shown in Figures 1 and 4, are provided relative to the infinite-demand throughput of 336 Mb/s, which we define as a relative load level of 1.0. The results, at a gross level, are similar to a standard FCFS discipline, which is surprising since the server attempts to process all requests fairly. No requests are intentionally delayed until others complete, and relatively small socket buffer sizes ensure that the largest requests must receive service piecemeal.

We notice that the main server process in Flash occasionally blocks, although it is designed to never do so. Then we instrument the kernel to discover the causes. By instrumenting the system call entry and exit points, the trap handler, and the scheduler, we are able to record the activity leading to the unexpected blocking.

Most blocking occurs in simultaneous access to metadata locks that are shared between the main Flash process (responsible for handling in-memory requests) and the helper processes (responsible for disk interactions). When this occurs, all in-memory requests being multiplexed by the main process are affected, leading to a breakdown in the effective service discipline. Generally speaking, less popular large files cause the processing of small files to be delayed, leading to sharp increases in latency.

To avoid this problem, we have the helper processes return open file descriptors using the `sendmsg()`, thereby eliminating any direct metadata access in the main process. A second problem, blocking in the `sendfile()` system call, is addressed by a combination of kernel optimization and small application modifications.

¹Details of the workloads and results for Apache are provided in our technical report [5]

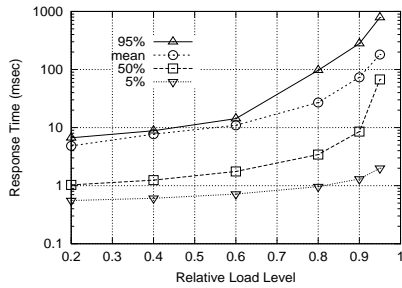


Figure 1: Latency distributions (mean, median, etc) for the Flash server at various load levels, where infinite demand (load level 1.0) = 336 Mb/s.

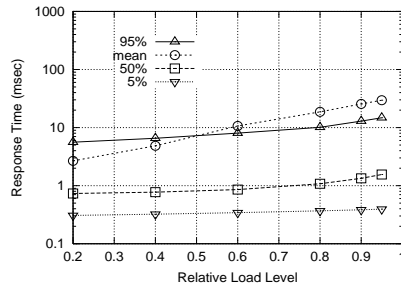


Figure 2: New-Flash – A factor of 6 in mean and 43 in median latency reduction, and 95% of requests are served in less than 15 ms. Infinite demand (load level 1.0) = 450 Mb/s.

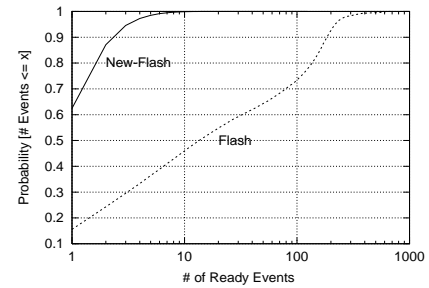


Figure 3: Event queue length for Flash variants – The original server has a mean of 61 events per dispatch, while the mean is only 1.6 in New-Flash.

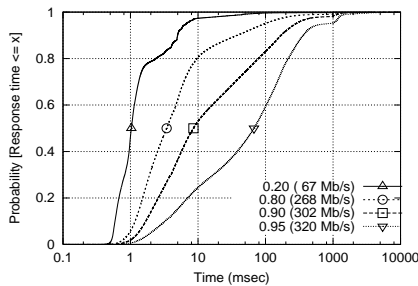


Figure 4: Flash latency CDF – Both median and mean latency grow significantly with load, though most requests should be served from memory

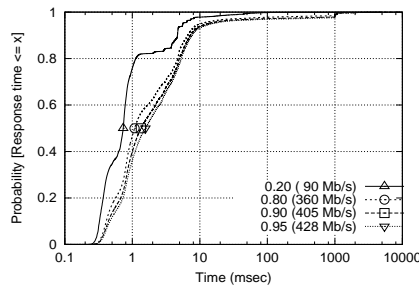


Figure 5: New-Flash latency CDF – Despite running at a higher request rate, base latency is lower, so is growth in latency with increasing load.

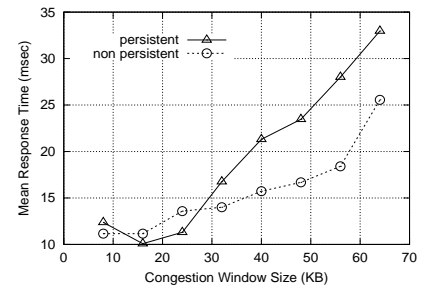


Figure 6: Mean latency vs. max congestion window size – Reducing the congestion window limit reduces bandwidth unfairness, and improves mean latency.

3. MAIN RESULTS

Latency Improvements: Our modified server, New-Flash, shown in Figures 2 and 5, shows about an order of magnitude reduction in mean latencies, and four to five orders reduction in the median and 95th percentile. The results are also *qualitatively* different – under heavy load, the 95% latency is relatively flat, and the mean values are higher than the 95% values. This indicates that the growth in mean latency is dominated by the tail of the response times, and are much more in line with what one would expect from processor sharing under a heavy-tailed workload. New-Flash exhibits a change in the effective service discipline. These results demonstrate that most server latency stems from head-of-line blocking, rather than standard queuing delays.

Throughput Improvements: In addition to the latency improvements, New-Flash also shows capacity gains of 34% at infinite demand. While impressive, much of this is a by-product of eliminating blocking, rather than the source of the latency improvements. The load levels in Figures 1 and 2 are relative to the infinite-demand rates of the servers, to normalize the capacity improvement.

Connection Scheduling: Figure 3 shows the CDF of ready events queue returned by the event delivery system calls (`select()` or `kevent()`) on our servers at infinite demand. Eliminating blocking reduces the burstiness of event delivery, and mean queue length drops from 61 to 1.6. With connection scheduling, original Flash’s latency drops 20-40%, but shows qualitatively similar profiles. In contrast, New-Flash shows no improvement at all, and in fact, even a pessimistic scheduling of Longest Remaining Processing Time shows no effect. Short event queues are not unreasonable – instead of waiting for application-level service, the server performs most tasks in the interrupt-driven network code.

Congestion Window Effects: With a median transfer size of less than 10 KBytes, most Web transfers complete while still in the TCP slow-start phase. Long-lived transfers not only exit slow start, but can open large congestion windows, unfairly dominating available bandwidth. Persistent connections can cause a similar effect, since multiple transfers over a single connection will appear to TCP as a single large transfer. To mitigate this behavior, we limit the size of the TCP congestion window and repeat the latency experiments at a 0.95 load level using New-Flash. The resulting mean latencies are shown in Figure 6, and exhibit even more improvement. This simple approach reduces the mean latency from an already low 33 ms to as little as 10 ms, almost a factor of 20 lower than the original Flash. While more work is needed to adapt to wide-area conditions, we believe this approach has much potential.

4. REFERENCES

- [1] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proc. of the 2nd USENIX Symp. on Internet Technologies and Systems (USITS'99)*, Boulder, CO, Oct. 1999.
- [2] M. Harchol-Balter, B. Schroeder, M. Agrawal, and N. Bansal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems*, 21(2):207–233, May 2003.
- [3] J. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *USENIX 2002 Annual Technical Conference*, pages 103–114, Monterey, CA, June 2002.
- [4] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX 1999 Annual Technical Conference*, pages 199–212, Monterey, CA, June 1999.
- [5] Y. Ruan and V. S. Pai. The origins of network server latency & the myth of connection scheduling. Technical Report TR-694-04, Princeton University, 2004.
- [6] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. of the 18th ACM Symp. on Operating System Principles*, pages 230–243, Chateau Lake Louise, Banff, Canada, Oct. 2001.