Frequency Domain Analysis and Visualization of Web Server Performance

Marc Chiarini and Alva Couch
Tufts University

November 5, 2007

1 Introduction

After many years of study, performance tuning of web servers remains more of an art than a science. One is often reduced to guessing what will improve performance via trial and error. Compounding this difficulty, we find that simple textbook models of web server behavior do not adequately describe the relationships between caching, resources, and performance. In this study, we looked closely at web server performance with the goal of defining and understanding normal behavior; we took measurements of a minimally configured web server expecting to see textbook curves of response time. Instead, we observed complex curves that demonstrate in theory that transactions are not satisfied in a "memoryless" manner, and in practice seem to be related to caching behavior. We believe that even a cursory study of these curves may lead to better models of server normalcy and may allow the everyday sysadmin to more quickly analyze web server failures and performance deficiencies.

Part of the reason that our results differ so much from classical results is that we utilized a novel measurement strategy. Rather than relying upon server logs, we employed the Linux auditing subsystem, which is normally intended for security logging. The auditing system allowed us to log the time taken to satisfy each request, rather than just the time at which the request entered the system or arrived at the client. We were then able to visualize the frequency of response times to a request. This strategy revealed that even simple web servers whose behavior should be straightforward exhibit unforeseen and sometimes unexplained complexity of response.

2 Methodology

We took several steps to build an observational (external) model of web server behavior. First, we constructed the following environment on a single subnet:

- Server:
  - RHEL 4 Linux on 2.4Ghz 32-bit PC with 1G of RAM, 1G swap, dual IDE hard drives in a software RAID-1 configuration, 100Mbps interface.
  - Out-of-the-box Redhat httpd (Apache 2.0.52) configured without caching
  - Linux audit subsystem records every TCP socket open and close at millisecond resolution.
  - All non-essential processes were halted. The server only ran kernel processes, httpd, xinitd, sshd, 2 mingetty, and a single xterm
- Clients:
  - 64-bit RHEL 4 SMP on multiple 2 and 4-CPU AMD Opteron machines with 4G of RAM.
  - Apache JMeter 2.3, a Java desktop application designed to load test functional behavior and measure performance. For our experiments, the software was only used to generate loads.

Next, we designed software in C and PERL to collect and post-process statistics about incoming HTTP requests generated by JMeter, as well as how the server responds in various dimensions, such as load average, CPU, memory, network, and disk utilization. We note that in any procedure of this sort, care must be taken to minimize errors resulting from sampling skew and quantization.

Although the web server was configured not to cache file requests, the operating system and the disk subsystem maintain their own caches. We wondered how much these cache sizes affect server performance, and hypothesized that the dominant effect causing the aforementioned response curves was caching. It was not technically feasible for us to disable the kernel and disk caches, so we tested our hypothesis instead via a "cache avoidance" technique. This consisted of reproducing the requested files in numbers that were sufficient to force the OS to retrieve them from disk on every request. Cache avoidance resulted in very different curves, much closer to theoretical performance models, so we conclude that kernel caching was a dominant factor in the prior results. It remains to be seen how much the audit subsystem and statistics collector, which both dirty these caches, might affect this scheme.

3 Testing

Web servers receive requests of all different shapes and sizes: images, documents and static HTML, dynamic HTML, Java Server Pages, and pages that require additional information from databases. To speak efficiently about web server inputs, we think about kinds of inputs as belonging to *classes* that utilize different amounts and sometimes different kinds of server resources. A class of inputs is a set of requests that have roughly the same effect upon the server and its resources when processed. Each class may additionally have subclasses that utilize server resources to differing extents. Our tests focused on static files of varying length; all files of the same length were in the same class.

To test single-class behavior under varying load conditions, we configured JMeter to use multiple threads to issue HTTP GET requests of a single 137K file from the server. In order to distinguish between steady-state behavior, and behavior at saturation, we increased the number of concurrent threads by a factor of two until we started observing erratic behavior.

To test multi-class behavior, we configured JMeter to issue equal numbers of HTTP GET requests of a small file (˜22K) and a large file (˜1.1M). We also experimented with test plans that requested files in four different classes (by size) and forty or more classes.

4 Results

Figure 1 shows a histogram of socket durations over a 60 minute period with 64 threads asking for the same file over and over. A majority of requests are completed between 450 ms and 600 ms and in fact, the mean response time is around 510 ms. The other curve shows the nearly ideal histogram that one would see in a perfect world: the number of requests that take longer and longer to satisfy decreases exponentially. Why do these two curves differ so greatly? We believe the dominant factor is caching. Files are served out of cache much more quickly than when they come off a disk. Caching changes the "normal" time it would take to service a request. In an ideal situation, one would expect caching to shorten all service times, but in practice this seems to be rarely the case.

When we try to avoid caching, we get the graph in Figure 2. This result comes closer to the ideal, but still fails to reach it. Most of the differences in this case seem to come from rotational latency on the disk as it quickly serves one file after the other. We actually get a much more even distribution of response times, subject to the layout of the files on disk and the disk drive's capabilities.

Figures 3 and 4 show a comparison of histograms for loads generated by exponentially increasing numbers of threads. In 3, we can see that the server response time starts to increase dramatically somewhere between 64 and 128 threads. The mean response time doubles between these two loads, as it does between all previous loads, but then the mean stops doubling and only grows by a constant factor. This tells us that beyond a 64 thread load, the server is reaching a saturation point, and further statistics will not be very reliable. The same result is obtained in the cache-avoidance tests (Figure 4), but it is harder to see with the naked eye. Generally, the greater the load, the farther away we get from an ideal server response. This brings up

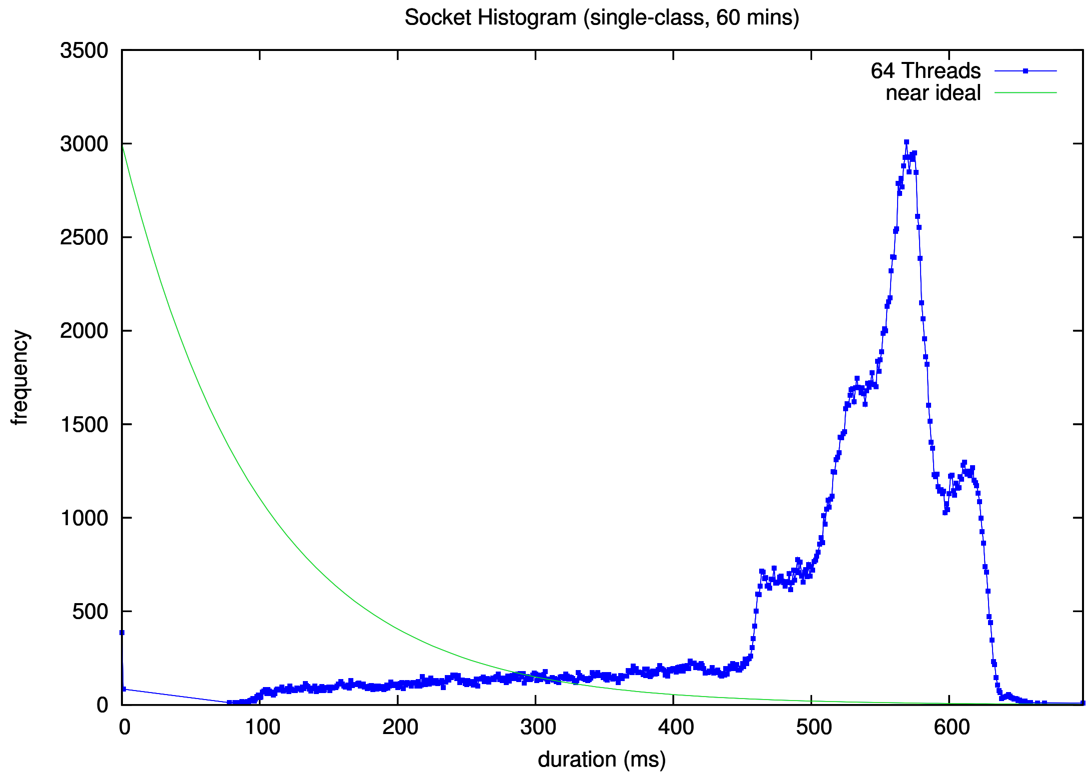

Figure 1: The y-axis shows how many sockets stayed open for the duration shown on the x-axis. The green curve represents an estimate of the ideal response.

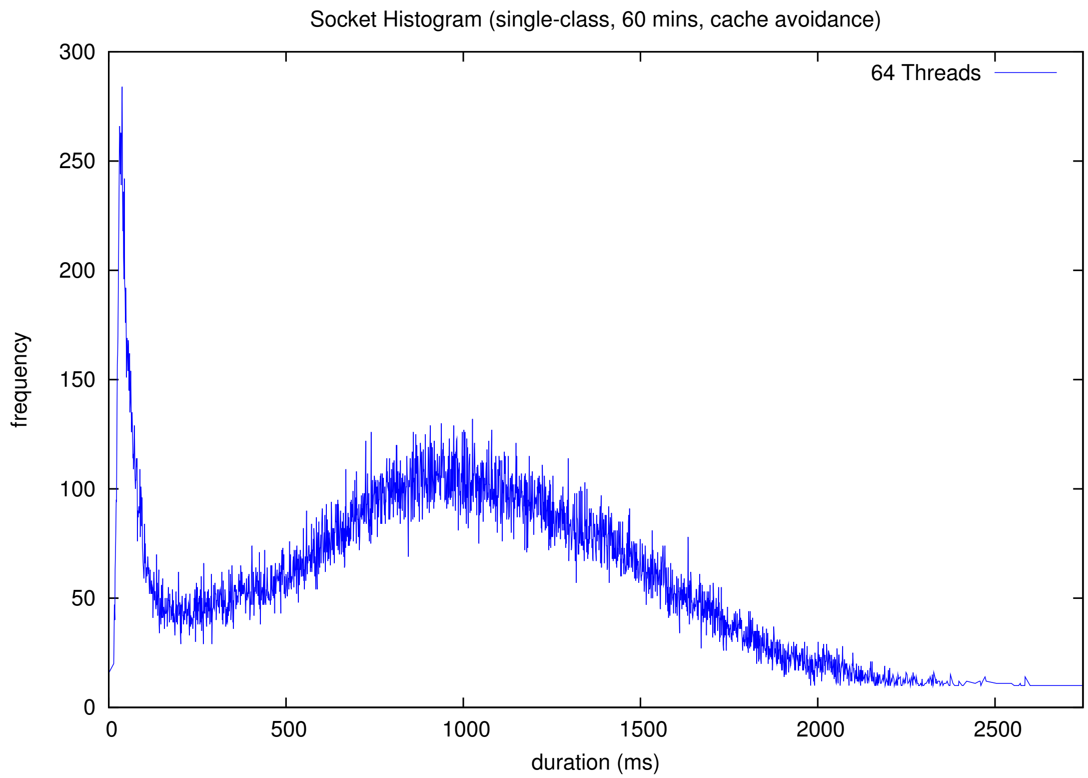

Figure 2: With cache avoidance, response times are more evenly distributed.

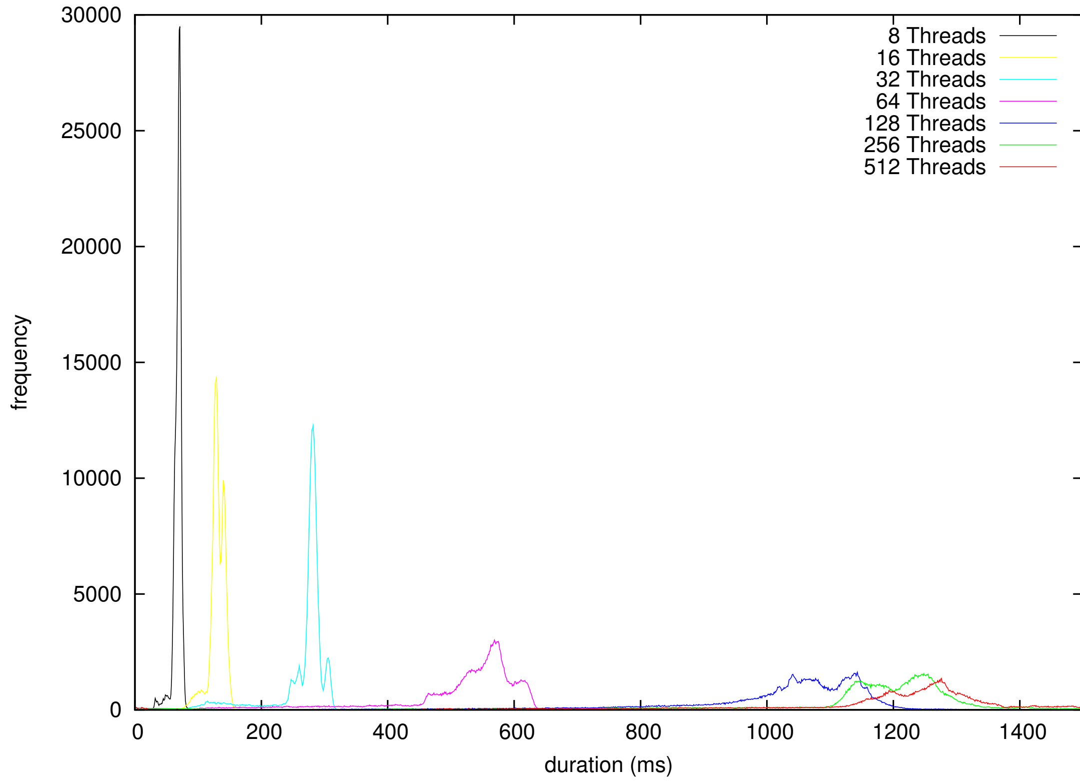

Figure 3: A comparison of 60-minute socket frequency histograms for a single request class over increasing loads.

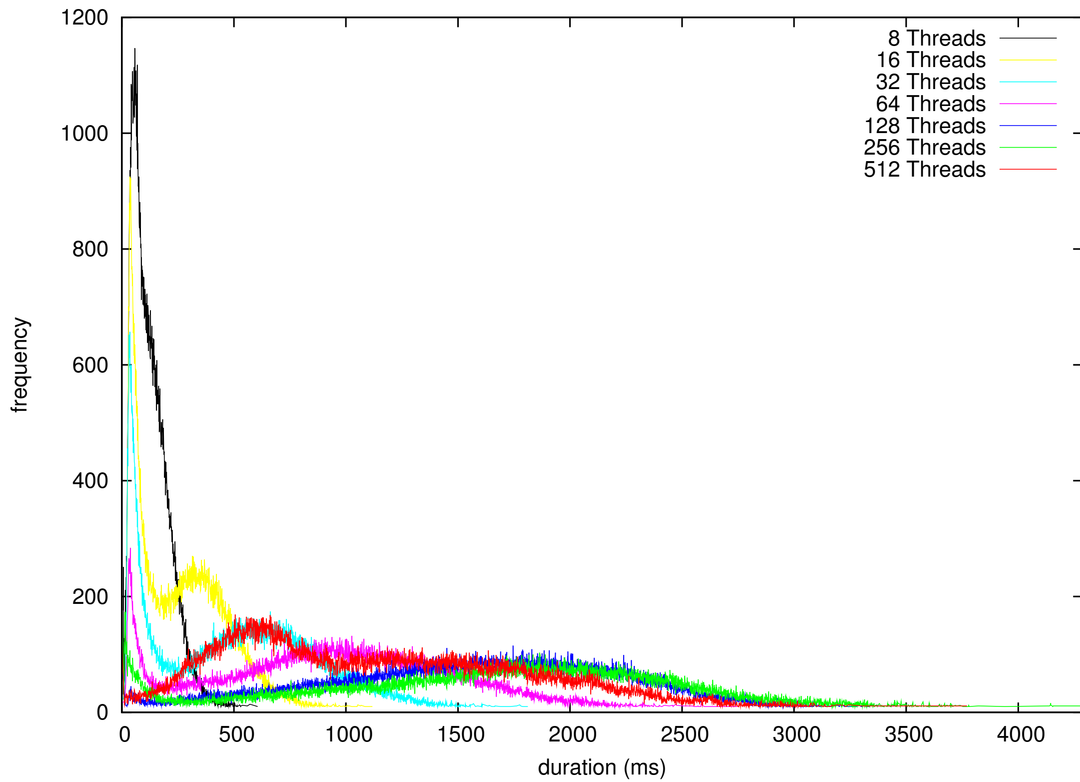

Figure 4: The same comparison as in Figure 3 but using cache-avoidance.

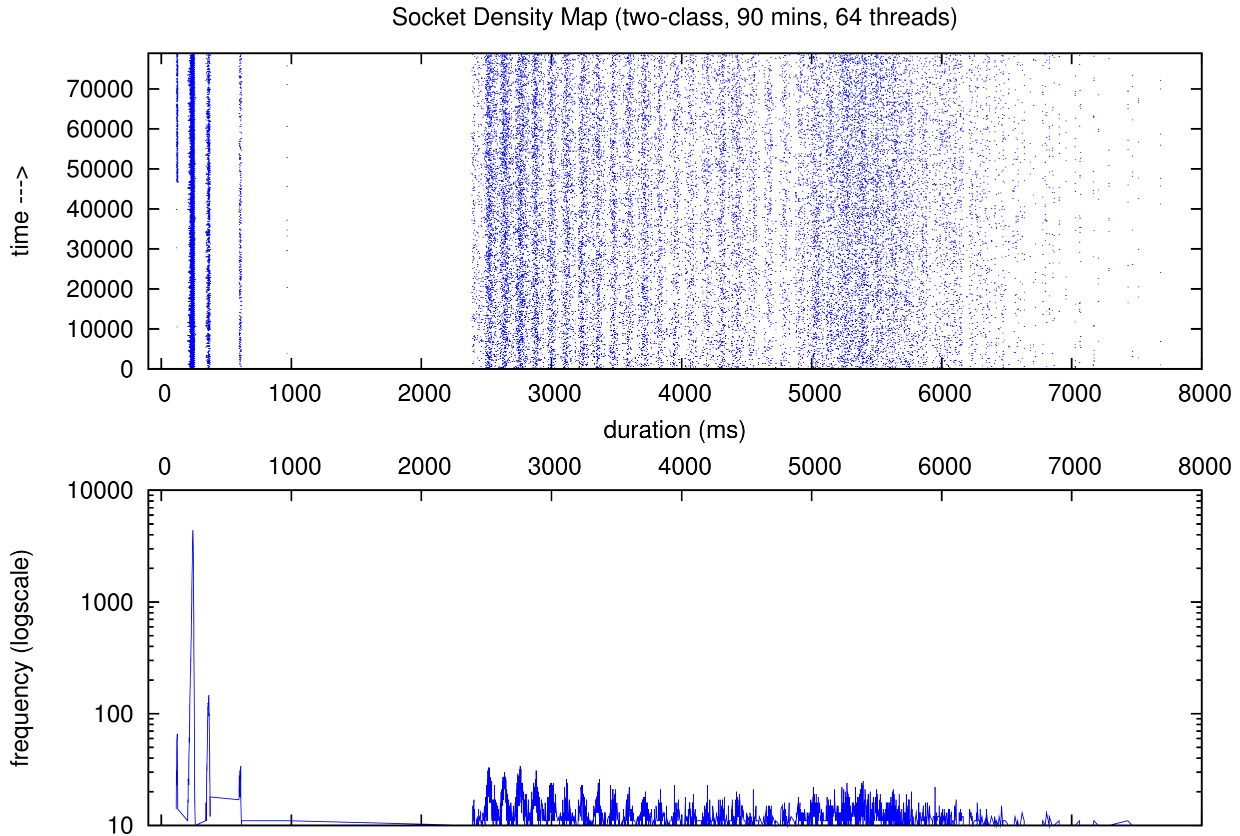

Figure 5: The socket density map is slightly deceptive, as it appears that responses to the two different classes appear on the left and right. In fact, 50% of requests were completed in less than the median duration of 245 ms.

the question of whether we can find a "sweet spot" for responses to common server loads. If so, we might consider dropping or delaying requests that tend to push the curve out of that zone.

When we began to examine multi-class test cases, we discovered another useful way of viewing web server behavior and potential performance pitfalls. Figure 5 pairs a socket histogram with a socket density map, which is a graph of how long each individual request took to complete. Although at first glance it appears that responses to the two different classes are nicely separated, the map in its current form is deceptive. The histogram is intentionally drawn with a logarithmic frequency scale to point out the two highest density duration zones on the left side. These likely represent the server response to the two classes of files, with the highest peak (shorter duration) representing the small file. We believe that the range of peaks in the histogram starting at around 2500 ms is related to cache flushes instigated by the measurement programs.

5 Conclusions

This research has introduced a new, practical method for studying web server behavior that relies on something at which humans excel: visual pattern recognition. Visualizations in the frequency domain enabled us to efficiently view the large differences between ideal behavior predicted by theoretical models, and actual behavior. With further refinement, we believe that a tool could be built (perhaps as a Nagios plug-in) that would help system and network administrators construct a useful picture of healthy server performance.

Our future work will include investigating how the "normal" graphs change in response to non-catastrophic failures and in response to new request classes. We would also like to study the response curves of server clusters that provide web service.