httperf—A Tool for Measuring Web Server Performance

David Mosberger
HP Research Labs
davidm@hpl.hp.com, http://www.hpl.hp.com/personal/David_Mosberger/

Tai Jin
HP Research Labs
tai@hpl.hp.com, http://www.hpl.hp.com/personal/Tai_Jin/

Abstract

This paper describes httperf, a tool for measuring web server performance. It provides a flexible facility for generating various HTTP workloads and for measuring server performance. The focus of httperf is not on implementing one particular benchmark but on providing a robust, high-performance tool that facilitates the construction of both micro- and macro-level benchmarks. The three distinguishing characteristics of httperf are its robustness, which includes the ability to generate and sustain server overload, support for the HTTP/1.1 protocol, and its extensibility to new workload generators and performance measurements. In addition to reporting on the design and implementation of httperf this paper also discusses some of the experiences and insights gained while realizing this tool.

1 Introduction

A web system consists of a web server, a number of clients, and a network that connects the clients to the server. The protocol used to communicate between the client and server is HTTP [2]. In order to measure server performance in such a system it is necessary to run a tool on the clients that generates a specific HTTP workload. Currently, web server measurements are conducted using benchmarks such as SPECweb or WebStone [6, 8] which simulate a fixed number of clients. Given that the potential user base of an Internet-based server is on the order of hundreds of millions of users it is clear that simulating a fixed and relatively small number of clients is often insufficient. For this reason, Banga and Druschel [1] recently argued the case for measuring web servers with tools that can generate and sustain overload, which is effectively equivalent to simulating an infinite user base. They also presented a tool called "s-clients" that is capable of generating such loads. The s-clients approach is similar to httperf in that both are capable of sustaining overload but they differ significantly in design and implementation. For example, httperf completely separates the issue of how to perform HTTP calls from issues such as what kind of workload and measurements should be used. Consequently, httperf can readily be used to perform various kinds of web-server related measurements, including SPECweb/WebStone-like measurements, s-client-like measurements, or new kinds of measurements such as the session-based measurements that we will discuss briefly in Section 4.

Creating httperf turned out to be a surprisingly difficult task due to factors inherent in the problem and shortcomings of current operating systems (OSes). The first challenge is that a web system is a distributed system and as such is inherently more difficult to evaluate than a centralized system that has little or no concurrency and a synchronized clock.

Second, HTTP in general and HTTP/1.0 in particular cause connection usage patterns that TCP was not designed for. Some of these problems have been fixed in response to the experience gained from running web servers. However, since tools such as httperf run on the client-side they exercise the system in a manner that is quite different from the way servers do. As a consequence, there are a number of additional issues that such a test tool needs to guard against.

A third challenge is that the world-wide web is a highly dynamic system. Almost every part in it—server and client software, network infrastructure, web content, and user access pattern—is subject to relatively frequent and fundamental changes. For a test tool to remain useful over a period of time requires a design that makes it relatively easy to extend and modify as need arises.

The rest of this paper is organized as follows: the next section gives a brief introduction on how to use httperf. Section 3 describes the overall design of the tool and presents the rationale for the most important design choices. Section 4 discusses the current state of httperf and some of the more subtle implementation issues discovered so far. Finally, Section 5 presents some concluding remarks.

2 An Example of Using httperf

To convey a concrete feeling of how httperf is used, this section presents a brief example of how to measure the request throughput of a web server. The simplest way to achieve this is to send requests to the server at a fixed rate and to measure the rate at which replies arrive. Running the test several times and with monotonically increasing request rates, one would expect to see the reply rate level off when the server becomes saturated, i.e., when it is operating at its full capacity.

To execute such a test, it is necessary to invoke httperf on the client machines. Ideally, the tool should be invoked simultaneously on all clients, but as long as the test runs for several minutes, startup differences in the range of seconds do not cause significant errors in the end result. A sample command line is shown below:

```
httperf --server hostname \
 --port 80 --uri /test.html \
 --rate 150 --num-conn 27000 \
 --num-call 1 --timeout 5
```

This command causes httperf to use the web server on the host with IP name *hostname*, running at port 80. The web page being retrieved is "`/test.html`" and, in this simple test, the same page is retrieved repeatedly. The rate at which requests are issued is 150 per second. The test involves initiating a total of 27,000 TCP connections and on each connection one HTTP call is performed (a call consists of sending a request and receiving a reply). The timeout option selects the number of seconds that the client is willing to wait to hear back from the server. If this timeout expires, the tool considers the corresponding call to have failed. Note that with a total of 27,000 connections and a rate of 150 per second, the total test duration will be approximately 180 seconds, independent of what load the server can actually sustain.

Once a test finishes, several statistics are printed. An example output of httperf is shown in Figure 1. The figure shows that there are six groups of statistics, separated by blank lines. The groups consist of overall results, results pertaining to the TCP connections, results for the requests that were sent, results for the replies that were received, CPU and network utilization figures, as well as a summary of the errors that occurred (timeout errors are common when the server is overloaded).

A typical performance graph that can be obtained with the statistics reported by httperf is shown in Figure 2. For this particular example, the server consisted of Apache 1.3b2 running on a HP NetServer with one 200MHz P6 processor. The server OS was Linux v2.1.86. The network consisted of a 100baseT Ethernet and there were four client machines running HP-UX 10.20. As the top-most graph shows, the achieved throughput increases linearly with offered load until the server starts to be-

```
Total: connections 27000 requests 26701 replies 26701 test-duration 179.996 s

Connection rate: 150.0 conn/s (6.7 ms/conn, <=47 concurrent connections)
Connection time [ms]: min 1.1 avg 5.0 max 315.0 median 2.5 stddev 13.0
Connection time [ms]: connect 0.3

Request rate: 148.3 req/s (6.7 ms/req)
Request size [B]: 72.0

Reply rate [replies/s]: min 139.8 avg 148.3 max 150.3 stddev 2.7 (36 samples)
Reply time [ms]: response 4.6 transfer 0.0
Reply size [B]: header 222.0 content 1024.0 footer 0.0 (total 1246.0)
Reply status: 1xx=0 2xx=26701 3xx=0 4xx=0 5xx=0

CPU time [s]: user 55.31 system 124.41 (user 30.7% system 69.1% total 99.8%)
Net I/O: 190.9 KB/s (1.6*10^6 bps)

Errors: total 299 client-timo 299 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

Figure 1: Example of Basic Performance Statistics

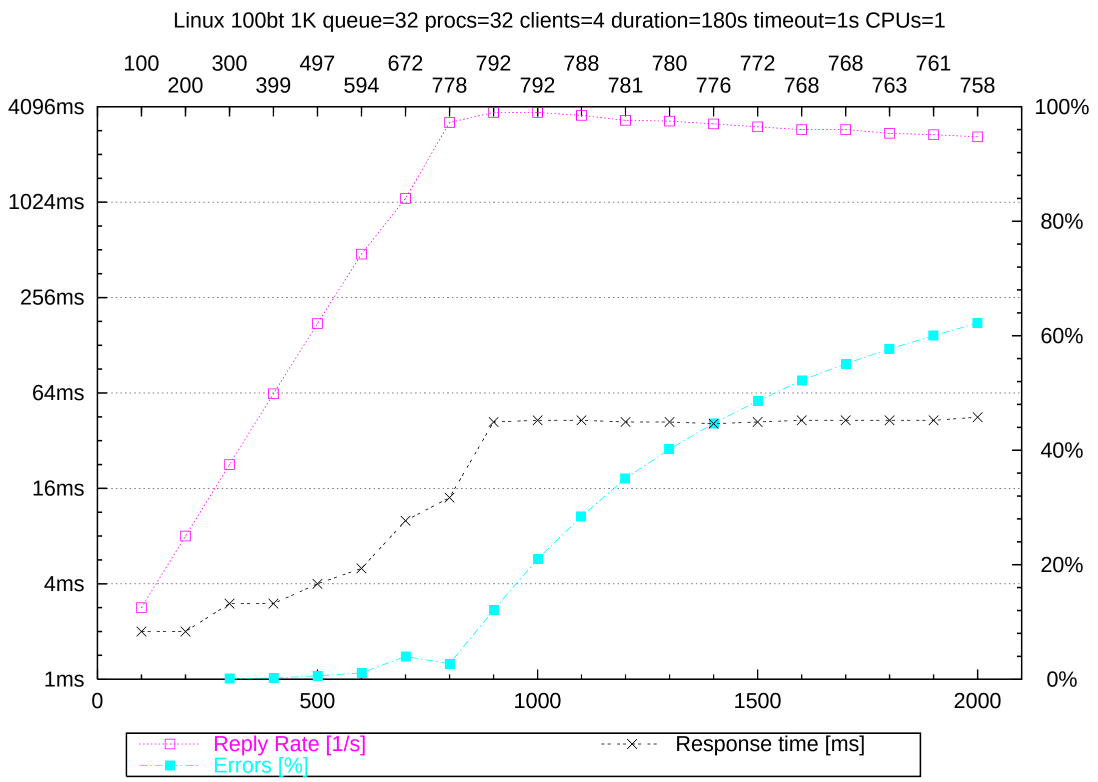

Figure 2: Example Server Performance Graph

come saturated at a load of 800 calls per second. As offered load is increased beyond that point, server throughput starts to fall off slightly as an increasing amount of time is spent in the kernel to handle network packets for calls that will fail eventually (due to client timeouts). This is also reflected in the error graph, which shows the percentage of calls that failed: once the server is saturated, the number of calls that fail increases quickly as more and more calls experience excessive delays. The third and final graph in the figure shows the average response time for successful calls. The graph shows that response time starts out at about 2ms and then gradually increases until the server becomes saturated. Beyond that point, response time for successful calls remains largely constant at 43ms per call.

3 Design

The two main design goals of httperf were (a) predictable and good performance and (b) ease of extensibility. Good performance is achieved by implementing the tool in C and paying attention to the performance critical execution paths. Predictability is improved by relying as little as possible on the underlying OS. For example, httperf is designed to run as a single-threaded process using non-blocking I/O to communicate with the server and with one process per client machine. With this approach, CPU scheduling is trivial for the OS which minimizes the risk of excessive context switching and poor scheduling decisions. Another example is timeout management: rather than depending on OS-mechanisms, httperf implements its own, specialized and light-weight timer management facility that avoids expensive system calls and POSIX signal delivery wherever possible.

Based on experiences with an earlier test tool, it was clear that httperf will undergo fairly extensive changes during its lifetime. To accommodate this need for change, httperf is logically divided into three different parts: the core HTTP engine, workload generation, and statistics collection. The HTTP engine handles all communication with the server and as such takes care of connection management and HTTP request generation and reply handling. Workload generation is responsible for initiating appropriate HTTP calls at the appropriate times so a particular workload is induced on the server. The third part, statistics collection, is responsible for measuring various quantities and producing relevant performance statistics. Interactions between these three parts occur through a simple yet general event signalling mechanism. The idea is that whenever something interesting occurs inside httperf, an event is signalled. Parties interested in observing a particular event can register a handler for the event. These handlers are invoked whenever the event is signalled. For example, the basic statistics collector measures the time it takes to establish a TCP connection by registering events handler for the events that signal the initiation and establishment of a connection, respectively. Similarly, a workload generator responsible for generating a particular URL access pattern can register a handler for the event indicating the creation of a new call. Whenever this handler gets invoked, the URL generator can insert the appropriate URL into the call without having to concern itself with the other aspects of call creation and handling.

3.1 Sustaining Overload

As alluded to earlier, an important design issue is how to sustain an offered load that exceed the capacity of the web server. The problem is that once the offered rate exceeds the server's capacity, the client starts building up resources at a rate that is proportional to the difference between offered and sustained rate. Since each client has only a finite amount of resources available, sooner or later the client would run out of resources and therefore be unable to generate any new requests. For example, suppose that each httperf process can have at most 2,000 TCP connection open at any given time. If the difference between offered and sustained rate is 100 requests per second, a test could last at most 20 seconds. Since web server tests usually require minutes to reach a stable state, such short test durations are unacceptable. To solve this problem, httperf times out calls that have been waiting for a server response for too long. The length of this timeout can be selected through command-line options.

With this timeout approach, the amount of client resources used up by httperf is bounded by the timeout value. In the worst case scenario where the server does not respond at all, httperf will never use more than the amount of resources consumed while running httperf for the duration of the timeout value. For example, if connections are initiated at a rate of 100 per second and the timeout is 5 seconds, at most 500 connections would be in use at any given time.

3.1.1 Limits to Client-Sustainable Load

It is interesting to consider just what exactly limits the offered load a client can sustain. Apart from the obvious limit that the client's CPU imposes, there is a surprising variety of resources that can become the first-order bottleneck. It is important to keep these limits in mind so as to avoid the pitfall of mistaking *client* performance limits as *server* performance limits. The three most important client bottlenecks are described below.

Size of TCP port space: TCP port numbers are 16 bits wide. Of the 64K available port numbers, 1,024 are typically reserved for privileged processes. This means that a client machine running httperf can make use of at most 64,512 port numbers. Since a given port number cannot be reused until the TCP TIME_WAIT state expires, this can seriously limit the client sustainable offered rate. Specifically, with a 1 minute timeout (common for BSD-derived OSes) the maximum sustainable rate per client is about 1,075 requests per second. With the RFC-793 [5] recommended value of 4 minutes, the maximum rate would drop to just 268 requests per second.

Number of open file descriptors: Most operating systems limit both the *total* and *per-process* number of file descriptors that can be opened. The system-wide number of open files is normally not a limiting factor and hence we will focus on the latter. Typical per-process limits are in the range from 256 to 2,048. Since a file descriptor can be reused as soon as an earlier descriptor has been closed, the TCP TIME_WAIT state plays no role here. Instead, the duration that is of interest here is the httperf timeout value. With a value of 5 seconds and assuming a limit of 2,000 open file descriptors per process, a maximum rate of about 400 requests per second could be sustained. If this becomes the first-order bottleneck in a client, it is possible to avoid it either by tuning the OS to allow a larger number of open file descriptors or by decreasing the httperf timeout value. Note that decreasing the timeout value effectively truncates the lifetime distribution of TCP connections. This effect has to be taken into consideration when selecting a timeout value.

Another seemingly obvious solution to work around the per-process limit on file descriptors would be to run multiple processes on a single machine. However, as will be explained in Section 4.1, there are other reasons that make this approach undesirable.

Socket buffer memory: Each TCP connection contains a socket receive and send buffer. By default, httperf limits send buffers to 4KB and receive buffers to 16KB. With limits in the kilobyte range, these buffers are typically the dominant per-connection costs as far as httperf memory consumption is concerned. The offered load that a client can sustain is therefore also limited by how much memory is available for socket buffers. For example, with 40MB available for socket buffers, a client could sustain at most 2,048 concurrent TCP connections (assuming a worst-case scenario where all send and receive buffers are full). This limit is rarely encountered, but for memory-constrained clients, httperf supports options to select smaller limits for the send- and receive-buffers.

The above list of potential client performance bottlenecks is of course by no means exhaustive. For example, older OSes often exhibit poor performance when faced with several hundred concurrent TCP connections. Since it is often difficult to predict the exact rate at which a client will start to become the performance bottleneck, it is essential to empirically verify that observed performance is indeed a reflection of the server's capacity and not that of the client's. A safe way to achieve this is to vary the number of test clients, making sure that the observed performance is independent of the number of client machines that participate in the test.

3.2 Measuring Throughput

Conceptually, measuring throughput is simple: issue a certain number of requests, count the number of replies received and divide that number by the time it took to complete the test. This approach has unfortunately two problems: first, to get a quantitative idea of the robustness of a particular measurement, it is necessary to run the same test several times. Since each test run is likely to take several minutes, a fair amount of time has to be spent to obtain just a single data point. Equally important, computing only one throughput estimate for the entire test hides variations that may occur at time scales shorter than that of the entire test. For these reasons, httperf samples the reply throughput once every five seconds. The throughput samples can optionally be printed in addition to the usual statistics. This allows observing throughput during all phases of a test. Also, with a sample period of 5 seconds, running a test for at least 3 minutes results in enough throughput samples that confidence intervals can be computed without having to make assumptions on the distribution of the samples [3].

4 Implementation

In this section, we first present the capabilities of the current version of httperf and then we discuss some of the more subtle implementation issues discovered so far. In the third part, we mention some possible future directions for httperf.

The HTTP core engine in httperf currently supports both HTTP/1.0 and HTTP/1.1. Among the more interesting features of this engine is its support for persistent connections, request pipelining, and the "chunked" transfer-encoding [2, 4]. Higher-level HTTP processing is enabled by the fact that the engine exposes each reply header-line and all of the reply body to the other parts of httperf by signalling appropriate events. For example, when one of the workload generators required simple cookie support, the necessary changes were implemented and tested in a matter of hours.

The current version of httperf supports two kinds of workload generators: request generators and URL generators:

Request Generation: Request generators initiate HTTP calls at the appropriate times. At present, there are two such generators: the first one generates new connections deterministically and at a fixed rate and each connection is used to perform a command-line specified number of pipelined HTTP calls. By default, the number of pipelined calls per connection is one, which yields HTTP/1.0-like behavior in the sense that each connection is used for a single call and is closed afterwards.

The second request generator creates *sessions* deterministically and at a fixed rate. Each session consists of a specified number of call-bursts that are spaced out by the command-line specified *user think-time*. Each call-burst consists of a fixed number of calls. Call-bursts mimic the typical browser behavior where a user clicks on a link which causes the browser to first request the selected HTML page and then the objects embedded in it.

URL Generation: URL generators create the desired sequence of URLs that should be accessed on the server. The most primitive generator simply generates the same, command-line specified URL over and over again.

The second generator walks through a fixed set of URLs at a given rate. With this generator, the web pages are assumed to be organized as a 10ary directory tree (each directory contains up to ten files or sub-directories) on the server. This generator is useful, for example, to induce a specific file buffer cache miss rate on the server under test.

As far as statistics collectors are concerned, httperf always collects and prints the basic information shown in Figure 1. The only other statistics collector at this time is one that collects session-related information. It measures similar quantities as the basic connection statistics with the main difference being that the unit of measurement is the session instead of the connection.

We now proceed to discuss some of the implementation issues that conspire to raise the difficulty to write a robust high-performance test tool.

4.1 Scheduling Granularity

The process scheduling granularity of today's OSes is in the millisecond range. Some support one millisecond, but most use a timer tick of around 10 milliseconds. This often severely limits the accuracy with which a given workload can be generated. For example, with a timer tick of 10 milliseconds, deterministically generating a rate of 150 requests per second would have to be implemented by sending one request during even-numbered timer ticks and two requests during odd-numbered ticks. While the average rate is achieved, the bursts sent during the odd-number ticks could cause server-queue overflows that in turn could severely affect the observed behavior. This is not to say that measuring web servers with bursty traffic is a bad idea (quite the opposite is true), however, the problem here is that burstiness was introduced due to the OS, not because the tester requested it.

To avoid depending on OS scheduling granularity, httperf executes in a tight loop that checks for network I/O activity via *select()* and keeps track of real time via *gettimeofday()*. This means that httperf consumes all available CPU cycles (on a multiprocessor client, only one CPU will be kept busy in this way). This approach works fine because the only other important activity is the asynchronous receiving and processing of network packets. Since this activity executes as a (soft-) interrupt handler, no scheduling problem arises. However, executing in a tight loop does imply that only one httperf process can be run per client machine (per client CPU, to be more precise). It also means that care should be taken to avoid any unnecessary background tasks on the client machine while a test is in progress.

4.2 Limited Number of Ephemeral Ports

Many TCP implementations restrict the TCP ports available to sockets that are not bound to a specific local address to the so-called ephemeral ports [7]. Ephemeral ports are typically in the range from 1,024 to 5,000. This has the unfortunate effect that even moderate request rates may cause a test client to quickly run out of port numbers. For example, assuming a TIME_WAIT state duration of one minute, the maximum sustainable rate would be about 66 requests per second.

To work around this problem, httperf can optionally maintain its own bitmap of ports that it believes to be available. This solution is not ideal because the bitmap is not guaranteed to be accurate. In other words, a port may not be available, even though httperf thinks otherwise. This can cause additional system calls that could ordinarily be avoided. It is also suboptimal because it means that httperf duplicates information that the OS kernel has to maintain at any rate. While not optimal, the solution works well in practice.

A subtle issue in managing the bitmap is the order in which ports are allocated. In a first implementation, httperf reused the most recently freed port number as soon as possible (in order to minimize the number of ports consumed by httperf). This worked well as long as both the client and server machines were UNIX-based. Unfortunately, a TCP incompatibility between UNIX and NT breaks this solution. Briefly, the problem is that UNIX TCP implementations allow pre-empting the TIME_WAIT state if a new SYN segment arrives. In contrast, NT disallows such pre-emption. This has the effect that a UNIX client may consider it legitimate to reuse a given port at a time NT considers the old connection still to be in TIME_WAIT state. Thus, when the UNIX client attempts to create a new connection with the reused port number, NT will respond with a TCP RESET segment that causes the connection attempt to fail. In the case of httperf this had the effect of dramatically reducing the apparent throughput the NT server could sustain (half the packets failed with a "connection reset by peer" error). This problem is avoided in the current version of httperf by allocating ports in strict round-robin fashion.

4.3 Slow System Calls

A final issue with implementing httperf is that even on modern systems, some OS operations are relatively slow when dealing with several thousand TCP control blocks. The use of hash-tables to look up TCP control blocks for incoming network traffic is standard nowadays. However, it turns out that at least some BSD-derived systems still perform linear control block searches for the *bind()* and *connect()* system calls. This is unfortunate because in the case of httperf, these linear searches can easily use up eighty or more percent of its total execution time. This, once again, can severely limit the maximum load that a client can generate.

Fortunately, this is an issue only when running a test that causes httperf to close the TCP connection—as long as the server closes the connection, no problem occurs. Nevertheless, it would be better to avoid the problem altogether. Short of fixing the OS, the only workaround we have found so far is to change httperf so it closes connections by sending a RESET instead of going through the normal connection shutdown handshake. This workaround may be acceptable for certain cases, but should not be used in general. The reason is that closing a connection via a RESET may cause data corruption in future TCP connections or, more likely, can lead to needlessly tying up server resources. Also, a RESET artificially *lowers* the cost of closing a connection, which could lead to overestimating a server's capacity. With these reservation in mind, we observe in passing that at least one popular web browser (IE 4.01) appears to be closing connections in this manner.

4.4 Future Directions

In its current form httperf is already useful for performing several web server measurement tasks but its development has by no means come to a halt. Indeed, there are several features that are likely to be added. For example, we believe it would be useful to add a workload generator that attempts to mimic the real-world traffic patterns observed by web servers. To a first degree of approximation, this could be done by implementing a SPECweb-like workload generator. Another obvious and useful extension would be to modify httperf to allow log file based URL generation. Both of these extensions can be realized easily thanks to the event-oriented structure of httperf.

Another fruitful direction would be to modify httperf to make it easier to run tests with multiple clients. At present, it is the tester's responsibility to start httperf on each client machine and to collect and summarize the per-client results. A daemon-based approach where a single command line would control multiple clients could be a first step in this direction.

5 Conclusions

The experience gained from designing and implementing httperf clearly suggests that realizing a robust and high-performance tool for assessing web server performance is a non-trivial undertaking. Given the increasing importance of the web and the need for quantitative performance analysis, the importance of such tools will continue to increase as well. While httperf is certainly not a panacea, it has proven useful in a number of web-related measurements tasks and is believed to be flexible and performant enough that it could provide a solid foundation to realize macro-level benchmarks such as SPECweb. For those cases where httperf may not be the solution of choice, we hope that the experience and lessons reported in this paper will be helpful in avoiding the most common pitfalls in measuring web server performance.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. Our thanks also go to Rick Jones, Rich Friedrich, and Gita Gopal for reviewing and commenting on draft versions of this paper on extremely short notice.

Availability

The httperf tool is available in source code form and free of charge from the following URL:

```
ftp://ftp.hpl.hp.com/pub/httperf/
```

References

[1] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server. In *USENIX Symposium on Internet Technologies and Systems*, pages 61–71, Monterey, CA, December 1997. http://www.usenix.org/publications/library/proceedings/usits97/banga.html.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force, January 1997. ftp://ftp.internic.net/rfc/rfc2068.txt.

[3] Rai Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, New York, NY, 1991.

[4] H. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, Hakon W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of SIGCOMM '97 Symposium*, pages 155–166, Cannes, France, October 1997. Association of Computing Machinery. http://www.acm.org/sigcomm/sigcomm97/papers/p102.html.

[5] Jon Postel. *Transmission Control Protocol*. DARPA, September 1981. ftp://ftp.internic.net/rfc/rfc793.txt.

[6] SPEC. An explanation of the SPECweb96 benchmark, December 1996. http://www.specbench.org/osg/web96/webpaper.html.

[7] Richard W. Stevens. *TCP/IP Illustrated: The Protocols*, volume 1. Addison-Wesley, Reading, MA, 1994.

[8] Gene Trent and Mark Sake. WebSTONE: The first generation in HTTP server benchmarking, February 1995. http://www.mindcraft.com/webstone/paper.html.