Submitted for publication to ACM Computer Communication Review.

Performance Interactions Between P-HTTP and TCP Implementations*

John Heidemann
USC / Information Sciences Institute

November 4, 1996

Abstract

This document describes several performance problems resulting from interactions between implementations of persistent-HTTP (P-HTTP) and TCP. Two of these problems tie P-HTTP performance to TCP delayed-acknowledgments, thus adding up to 200ms to each P-HTTP transaction. A third results in multiple slow-starts per TCP connection. Unresolved, these problems result in P-HTTP transactions which are 14 times slower than standard HTTP and 20 times slower than potential P-HTTP over a 10 Mb/s Ethernet. We describe each problem and potential solutions. After implementing our solutions to two of the problems, we observe that P-HTTP performs better than HTTP on a local Ethernet. Although we observed these problems in specific implementations of HTTP and TCP (Apache-1.1b4 and SunOS 4.1.3, respectively), we believe that these problems occur more widely.

1 Introduction

At ISI we are currently examining HTTP protocol performance across various transport protocols [7, 15]. As a part of this work we have examined the performance of HTTP and persistent-HTTP (P-HTTP) in detail. We have developed a model for HTTP performance based on a function of server and network characteristics [7].

To validate our HTTP performance model we compared predicted performance to measured performance in an actual web server. Our early experiments suggested that P-HTTP performance was ten times slower than the corresponding HTTP transactions in a simple page-retrieval benchmark. This result is surprising since P-HTTP is intended to improve performance by amortizing costs of connection creation across multiple requests [11, 10].

We found several interactions between P-HTTP and TCP which explain the exceedingly poor P-HTTP performance. These performance problems are not caused by specific errors in our server (Apache, beta version 1.1b4) or in our TCP implementation (SunOS 4.1.3), but they instead result from interactions between application-level P-HTTP behavior and existing TCP algorithms. Resolution of these interactions provides an HTTP implementation where P-HTTP is 40% faster than simple HTTP over an Ethernet. With these implementation changes, most P-HTTP overhead is accurately accounted for by our analytic model [7].

Although we found problems in particular implementations of P-HTTP and TCP, we believe that there are several reasons broader understanding of these issues is needed in the web community. First, P-HTTP is a relatively new protocol and is only now becoming standardized [5]. Although P-HTTP is derived from HTTP, P-HTTP exhibits very different network dynamics. To a first approximation, simple HTTP is identical to the data channel of FTP: a new connection is opened for each data object. FTP behavior has been studied for many years. P-HTTP involves multiple exchanges over a single TCP connection, thus it behaves much more like

[*] This research is supported by the Defense Advanced Research Projects Agency (DARPA) through FBI contract J-FBI-95-185 entitled “Cities Online”. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of the Army, DARPA, or the U.S. Government.

The author can be contacted at 4676 Admiralty Way, Marina del Rey, CA, 90292-6695, or by electronic mail to johnh@isi.edu. Other information about the LSAM project can be found at http://www.isi.edu/lsam/.

SMTP or NNTP (the Internet's standard e-mail and news transfer protocols) than FTP. SMTP is a batch protocol and NNTP usually runs over LANs, so it is not surprising that TCP is not tuned for wide-area P-HTTP-style traffic.

Second, we have observed these problems in widely deployed implementations of HTTP and TCP. We have also made an early draft of this work available to others and been told that similar problems exist in at least one other HTTP server [6]. Together, these observations suggest that the web development community is not widely familiar with these problems.

Finally, HTTP is becoming very widely deployed outside its original domain of hypertext exchange. HTTP server implementations have been deployed for weather sensor arrays, networked disk drives, network routers and gateways, and implementations exist for nearly all types of general-purpose computers. Although many of these platforms will implement only a subset of HTTP (and possibly not P-HTTP), the many potential P-HTTP implementations suggest that a broader understanding of its behavior is important.

This document summarizes two observed performance problems and a third anticipated problem. In each case we describe the problem and demonstrate it with packet traces (where possible). We have implemented solutions to the first two problems we describe and show that, with these solutions, P-HTTP performs better than HTTP. We outline a solution to the third problem.

2 The Performance Problems

Of the three performance problems identified in our work, two involve delayed acknowledgments, and the third concerns congestion control. This section describes each problem and their solutions. Detailed fixes for the problems are available from our web page at `http://www.isi.edu/lsam/tools/`.

2.1 Experimental Framework and Initial Performance

Our experiments involved two hosts directly connected by a 10 Mb/s Ethernet (RTT less than 1ms, measured user-to-user bandwidth 8.7Mb/sec). These experiments were performed between two Sun SPARC 20/71 computers running SunOS 4.1.3 with some TCP modifications (multicast support, a 16KB default TCP window size, and slow-start enabled for directly connected networks). Our HTTP server was Apache 1.1b4 and the client was a custom program written in Perl. The client made HTTP/1.0 requests; persistent connections were indicated with a "Connection: Keep-Alive" header.

With this configuration we ran a workload consisting of 100 web-page transactions. Each retrieval consists of retrieval for three documents of 6,651, 3,883, and 1,866 bytes over a single P-HTTP connection. (These documents are the same size as the front page of Yahoo (`http://www.yahoo.edu/`) on May 1, 1996, and represent a hypertext document with two embedded images.)

Initial transaction times for HTTP and P-HTTP are summarized in the first two rows of Table 1. P-HTTP performance is about 14 times worse than simple HTTP performance over Ethernet. The relative overhead would be substantially less if these pages were accessed over wide-area networks with lower bandwidth and higher latencies. Nonetheless, this high overhead suggests that something is wrong with P-HTTP performance for these implementations.

As of this writing the current release of Apache-1.1.1 is available (at `http://www.apache.org/`). Although we have not repeated our experiments with this release, the code relevant to these problems does not appear to have changed. We have informed the Apache developers of the problems and fixes we discuss below; we expect that some of our patches will be part of a future Apache release.

2.2 The Short-Initial-Segment Problem

The first problem we encountered was an interaction between Apache sending MIME headers as a separate segment and SunOS' implementation of TCP's slow-start and delayed-acknowledgment algorithms.

Apache supports keep-alive connections, an early implementation of P-HTTP. When handling a keep-alive connection, Apache sends its headers as a separate segment. (It does so to work around a bug in a popular browser.) TCP MSS (maximum segment size) is typically 1460B for Ethernet or 512B or 536B for wide-area TCP connections. HTTP headers are much less than a full segment, typically 200–300 bytes. TCP's slow-start algorithm specifies that the connection opens its congestion window exponentially. For each segment acknowledged the congestion window increases by a full-size segment, allowing two segments to be introduced into the network (one replacing the old segment and one new segment).

When a server replies to an HTTP request the congestion window begins at two segments[1]; thus the Apache server will send one small segment with the HTTP headers followed by a second segment of size MSS. It then waits for an ACK before continuing.

[1] The HTTP-reply congestion window starts at two segments in most BSD-derived TCP implementations because the ACK of connection setup has already opened the congestion window by one segment.

	server	retrieval time
HTTP	1.1b4	43ms (4.0ms, 94%, ±8.0ms)
P-HTTP	1.1b4	605ms (10ms, 16%, ±19.7ms)
P-HTTP	1.1b4 (w/first fix)	195ms (1.9ms, 1%, ±0.38ms)
P-HTTP	1.1b4 (w/both fixes)	26ms (8.8ms, 33%, ±1.7ms)

Table 1: Retrieval time (for HTML and images) for different protocol and software versions. Each measurement is the average of 100 samples. Values in parentheses give standard deviation, percent relative standard deviation, and 95% confidence intervals.

The client reads both of these segments. TCP's delayed-acknowledgment algorithm specifies that ACKs should be delayed in hopes of piggybacking the ACK on return traffic. The host requirements RFC adds that at least every other full segment must be acknowledged [1]. Unfortunately, the client has received only one full segment and one partial segment. The client therefore delays ACKing the data until the delayed ACK timer expires, which can take up to 200ms on BSD-derived TCPs or 500ms according to the specification [1].

A packet trace illustrating this problem can be seen in Figure 1. Details of the packet exchanges are listed in Figure 2. Although this trace represents a single response, time between the second and third data segments is consistently 170–190ms in our experiments. After the first exchange, the client actually becomes synchronized with the server's slow-start clock.

A solution to this problem is to insure that the HTTP server does not send the HTTP headers in a partial segment. Apache sent the headers with an explicit application-level flush; removing this flush causes the headers to be sent with the initial data. This flush was explicitly added to Apache for persistent connections to work around a bug in a popular browser; we discuss this problem in Section 2.6.

Resolution of this problem improves P-HTTP performance substantially. The third row of Table 1 shows Apache performance with this fix. While P-HTTP performance reduced to a third of unmodified Apache's P-HTTP (the second row), it is still substantially worse than simple HTTP performance.

We believe that this problem is an example of a broader problem in using TCP for request–response traffic. TCP delays acknowledgments with the goals of piggy-backing them on return traffic and of reducing ACK frequency. For request–response usage (such as HTTP), piggy-backing is rarely successful since data traffic is almost completely unidirectional. This same problem occurs in FTP data traffic; each FTP data exchange stalls for up to the delayed-ACK time-out period.

With primarily unidirectional traffic segments are usually sent back-to-back. A better approach for such traffic would be to delay ACKs by the time required to send a full segment and then immediately send an ACK, thus consolidating every other ACKs without unnecessarily delay.

2.3 The Odd/Short-Final-Segment Problem

The second problem we encountered involved odd numbers of segments interacting with the silly-window-syndrome (SWS) avoidance algorithm [4]. The problem occurs when the Nagle algorithm is enabled and a response requires an odd number of full segments followed by a short final segment. The Nagle algorithm was designed for terminal I/O traffic and so is not appropriate for HTTP traffic, but it is enabled by default and has not been a problem with simple (non-persistent connection) HTTP traffic.

Odd numbers of segments arise when Apache sends data over a TCP connection with a large MSS. TCP connections between Ethernet-connected hosts typically have an MSS of 1460B, as would wide-area connections where the hosts implement MTU-discovery [9]. (Without MTU-discovery wide-area connections typically see a 512B or 536B MSS.)

Apache writes data at the application-layer in 4KB chunks. TCP breaks this data into three segments of lengths 1460, 1460, and 1175. The client will acknowledge the first two segments immediately upon receipt (recall that according to the host requirements RFC, every two full segments must be acknowledged [1]). The client will delay acknowledgment of the third segment according to the TCP delayed acknowledgment algorithm.

Next assume that the server has only a small amount of data to send to complete the current response (small here means less than half of the client's maximum advertised window). Apache will immediately write this data. TCP, however will refuse to send it because of sender-side SWS avoidance [3]. According to Stevens' summary of the BSD TCP algorithms [12] (paraphrased from page 326), the server won't send data until: (a) a full-size segment can be sent, (b) we can send half of the client's advertised window, (c) we can send everything we have and either are not expecting an ACK or the Nagle algorithm is disabled. Cases (a) and (b) can never be

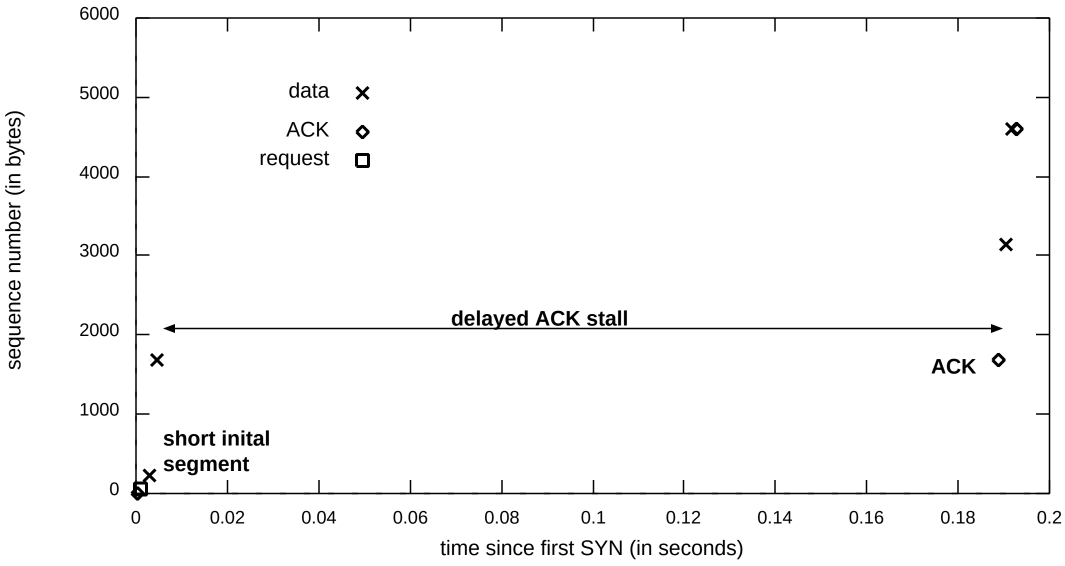

Figure 1: A sequence-number plot illustrating the short-initial-segment problem. The segment through byte 227 is short. It is followed by a long segment (through 1687B) and then stalls until the delayed ACK at 0.189201.

```
0.000000 client.3199 > server.8080: S 676352001:676352001(0) win 16384 <mss 1460>
0.000251 server.8080 > client.3199: S 1807168001:1807168001(0) ack 676352002 win 16384 <mss 1460>
0.000460 client.3199 > server.8080: . ack 1 win 16384
# request
# client_think_time: 0.650ms
0.001110 client.3199 > server.8080: P 1:53(52) ack 1 win 16384
# response
# server_think_time: 2.175ms
# first segment: headers
0.003285 server.8080 > client.3199: P 1:227(226) ack 53 win 16384
# second segment: data
# delta_time: 1.626ms
0.004911 server.8080 > client.3199: . 227:1687(1460) ack 53 win 16384
# server delays for ACK before sending next segment
0.189201 client.3199 > server.8080: . ack 1687 win 16384
# additional data
# delta_time: 185.736ms
0.190647 server.8080 > client.3199: . 1687:3147(1460) ack 53 win 16384
0.191879 server.8080 > client.3199: . 3147:4607(1460) ack 53 win 16384
```

Figure 2: Packet trace demonstrating the short-initial-segment problem. The segment through byte 227 is short. It is followed by a long segment and then stalls waiting on the delayed ACK.

true for the transaction because we're sending the last few bytes of the response. Case (c) is not true because we have outstanding unacknowledged data (the odd segment) and Nagle is enabled by default. The server therefore waits for the client to ACK this segment before responding. Delaying acknowledgments means that the client will not do so for up to 200ms.

This problem is illustrated graphically in Figure 3. The detailed packet trace of a portion of the plot is listed in Figure 4. Again, although this trace represents a single run, the time between the second and third data segments is consistently 170–190ms.

This problem occurs because Nagle's algorithm is intended for small-packet, interactive traffic while P-HTTP uses TCP for a series of requests and responses. This problem does not occur with non-persistent HTTP requests because closing the TCP connection also immediately sends any data waiting for transmission. We solve this problem by disabling Nagle's algorithm for P-HTTP connections, thus disabling the aspect of SWS avoidance which interferes with performance.

Resolution of this second problem brings P-HTTP performance in line with what we expect (see the final line of Table 1); P-HTTP performs better than simple HTTP by avoiding connection setup costs. With this fix we observe actual P-HTTP performance over wide-area connections that is with 5% of that predicted by our model of TCP connection setup behavior [7].

2.4 The Slow-Start Re-Start Problem

A final potential problem we are aware of involves conservative assumptions made in some TCP implementations about congestion control. These assumptions originated post-4.3BSD TCP [8] and do not occur in many BSD-derived systems (such as SunOS). The interaction between these assumptions and P-HTTP was originally observed in other work on P-HTTP performance [15].

BSD TCP makes a very conservative assumption about the congestion window. If at any time all data sent has been acknowledged and nothing has been sent for one retransmission time-out period, then it reinitializes the congestion window to 1 segment, forcing a slow-start. The motivation for this algorithm was the observation that some applications such as SMTP and NNTP typically have a negotiation phase followed by a data transfer phase [8]. The negotiation phase can artificially open the congestion window; data transfer will then result in a burst of packets which can move the network out of equilibrium, potentially resulting in congestion or packet loss.

A result of reinitializing the congestion window is that, even without packet loss, P-HTTP connections will frequently slow-start "mid-stream". In fact, since users nearly always spend more than the retransmit-ion time-out browsing a given page, P-HTTP will nearly always slow-start when the user follows a link. The primary goal of P-HTTP is to avoid the cost of multiple connection setup and slow-starts; this interaction defeats much of the purpose of P-HTTP's optimization. Web pages today typically require a "cluster" of HTTP requests, one for the HTML document and one for each embedded image. While P-HTTP's optimizations will be successful across a cluster, they will not be between clusters, thus limiting P-HTTP performance [7].

We have not yet experimentally verified that this behavior occurs. We have, however, examined the source code of several existing Unix implementations. SunOS 4.x does not reduce the congestion window except due to packet loss. 4.4BSD, FreeBSD 2.1, and Linux 2.0 will reset the congestion window. Stevens describes this behavior (Section 26.2, [13]), although he states that the idle time is one round-trip time rather than the retransmission time-out interval.

Several solutions exist to unify the goals of the TCP layer (congestion avoidance via packet conservation) and P-HTTP (maximum throughput). First, one could omit the code to reset the congestion window (as in SunOS 4.1.3) or significantly increase the time before the window is closed. This approach improves P-HTTP performance by avoiding additional slow-starts, but will send a burst of up to a full window of packets. In an internetwork, bursty traffic can result in packet loss due to router queue overflow, possibly resulting in poorer performance overall (both for the P-HTTP connection and for other traffic).

The other extreme is to insure that all TCP implementations reset the congestion window after an idle period. Ideally the window would be closed in the kernel as is done 4.4BSD and Linux. In addition, application-level protocols could implement this algorithm by closing connections after the appropriate length of time. Unfortunately, adjusting this time to network behavior requires information (the round-trip estimate) not easily available to the application. This approach also limits the performance advantage of persistent connections.

We believe that an intermediate approach is preferable to the alternatives. One intermediate approach would be to decay the congestion window over time rather than reset it to one. This approach improves P-HTTP performance but can still result in packet bursts unless the window is capped at some value. Determining parameters for this approach is difficult.

A preferable intermediate approach would be to keep the window open but to limit the rate at which packets are introduced into the network, thus avoiding burstiness. Although rate-based flow-control is difficult to use for new TCP connections because network conditions

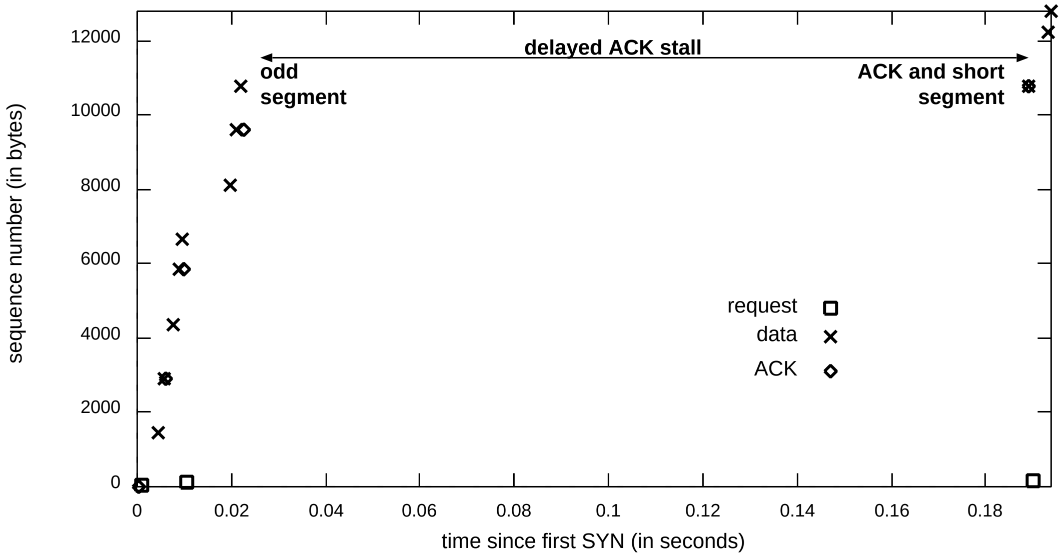

Figure 3: A sequence-number plot illustrating the odd/short-final-segment problem. The segment at 10772B is an odd segment, a short segment at 10785B is available for transmission but cannot be sent until the ACK arrives at 0.189262ms.

(connection setup and first P-HTTP request omitted)

```
% base time 9:44:07.871221  +10598
# request
0.010598 client.1029 > server.8080: P 53:111(58) ack 6676 win 16384
0.011122 server.8080 > client.1029: . ack 111 win 16326
# reply
0.019869 server.8080 > client.1029: . 6676:8136(1460) ack 111 win 16384
0.021099 server.8080 > client.1029: . 8136:9596(1460) ack 111 win 16384
0.022103 server.8080 > client.1029: P 9596:10772(1176) ack 111 win 16384
# ACK first two packets
# (ack was queued at client pending transmission of third packet)
0.022835 client.1029 > server.8080: . ack 9596 win 16384
# ACK is delayed 166ms
0.189262 client.1029 > server.8080: . ack 10772 win 16384
# final packet immediately follows ack
0.189479 server.8080 > client.1029: P 10772:10785(13) ack 111 win 16384
```

Figure 4: Portion of a packet trace demonstrating the odd/short-final-segment problem.

are unknown, TCP connections could collect congestion control information and apply it to limit data flow when restarting transmission after the connection goes idle. The TCP implementation might estimate packet rate by counting packets sent in a given packet's round trip. If n packets were sent in time t, packets after an idle connection would be sent out one every n/t seconds.

A rate-based algorithm for mid-stream re-starts provides a good balance between the desires for good HTTP performance and steady packet traffic. Solutions like this one can be deployed incrementally since for HTTP traffic only the server's TCP implementation need change. An open issue is how difficult it would be to add this algorithm to existing TCP implementations. We plan to explore this issue in future work, possibly using TCP-Vegas mechanisms to measure transfer rate [2] and adding a new rate-based alternative to slow-start.

Finally, the question of how to initialize or reset TCP status information over time and space arises not only when a connection goes idle, but also when initiating new connections in parallel or serial. For a more detailed discussion of the alternatives, see [14].

2.5 Other problems

In addition to interactions between P-HTTP and TCP we have observed two performance problems not specific to P-HTTP. These problems and their solutions have been widely explored; we describe them here briefly in the context of Apache.

First, many web servers employ standard I/O packages (for example, C's stdio library). The buffering used in these packages allows the application to do small I/Os efficiently (by merging several small, application writes into a single system-level write) but result in several extra data copies for bulk data transfer. For example, using stdio for both input and output requires up to six data copies (disk, file-system cache, input-stream stdio buffer, user buffer, output stdio buffer, network buffers, network device). The common solution to this problem is memory-mapping the input file, reducing data copies to three (disk, file-system cache, network buffers, network device). With memory-mapping all data copying can happen directly in the kernel.

Both Apache 1.0.5 and NCSA 1.5 use the C standard I/O library; Apache 1.1 replaces stdio with a custom library with a similar buffering scheme. We have modified Apache to use stdio for header output and switch to memory-mapped files for bulk-data transfer. To avoid the short-initial-segment problem we also write enough of the initial data to insure a steady return of ACKs. Since there is a fixed overhead in setting up memory-mapping we enable it only for files larger than 8KB.

The second general problem we encountered was socket buffers too small to support steady segment flow for wide-area connections. TCP's sliding-window is limited by socket buffer size; if this window is smaller than the bandwidth-delay product between the client and the server, the server will be unable to send enough data to keep the pipe full and performance will be less than optimal. The default size of a TCP socket buffer is system dependent, ranging from 2–16KB. A common value is 4KB. Although this may be sufficient for connections with a low bandwidth-delay product (such as modems and ISDN), well connected hosts will find it insufficient when browsing distant web pages. For example, well-connected hosts crossing the United States benefit from a 12KB or larger buffer (1Mb/sec bandwidth at 90ms latency). A default buffer size of 16KB is not unreasonable, larger buffers are recommended for particularly well-connected hosts.

Neither Apache nor NCSA set the socket send-buffer size. We have modified Apache to support a configurable send buffer.

2.6 Current Status

We have modified Apache to solve or work around each of these problems except for the slow-start re-start problem. We are currently exploring options and plan to modify TCP to address that problem shortly. The effects of our modifications are seen in Table 1. With the fixes, P-HTTP performance is better than non-persistent HTTP.

Our patches to Apache are available from our web page at `http://www.isi.edu/lsam/tools/`. We have discussed the problems we have observed with the Apache developers; our fixes to the odd/short-final-segment problem and the send-buffer-size problem will be in the next release.

Incorporation of our short-initial-segment fix has been discussed by the Apache developers. Code to flush headers (and thus send them as a separate segment) was added to Apache specifically to work around a bug in Netscape Navigator's implementation of persistent connections. Navigator appears to discard data sent after the headers in a persistent connection under some circumstances. We are currently investigating this bug. Until the Navigator bug is resolved, it appears best to either disabling persistent connections or keeping the explicit flush for clients which identify themselves as Netscape in their request headers.

3 Conclusions

We have identified three performance problems that occur due to interactions between specific implementations of TCP and P-HTTP. We have demonstrated that two of these interactions can result in P-HTTP performance 20 times slower than possible for hosts on a directly connected, 10 Mb/s Ethernet, and that the third can substantially reduce the performance benefits of P-HTTP.

Although our observations of these interactions are specific to BSD-derived TCPs and the first two are specific to the Apache HTTP server, these implementations are widely used. To avoid similar situations in other implementations, developers must be aware of these interactions. We have suggested solutions to each of the problems, and implemented solutions to the first two problems, demonstrating that these solutions bring Apache P-HTTP performance in line with expectations.

Acknowledgments

I would like to thank Katia Obraczka, Joe Touch, and Ted Faber for their discussions about these performance problems. I would also like to thank Rod van Meter, Joe Bannister, and Jon Postel for suggestions about this paper. Finally, I would like to thank the members of the Apache developers mailing list, particularly Randy Terbush, for their comments upon this work and these patches.

References

[1] R. Braden. Requirements for Internet hosts—communication layers. RFC 1122, Internet Request For Comments, October 1989.

[2] L. Brakmo and L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal of Selected Areas in Communication*, 13(8):1465–1480, October 1995.

[3] David D. Clark. Modularity and efficiency in protocol implementation. RFC 817, Internet Request For Comments, July 1982.

[4] David D. Clark. Window and acknowlegement strategy in TCP. RFC 813, Internet Request For Comments, July 1982.

[5] R. Fielding, H. Frystyk, T. Berners-Lee, J. Gettys, and J. Mogul. Hypertext transfer protocol—HTTP/1.1. RFC draft-ietf-http-v11-spec-04.txt, Internet Request For Comments, June 1996.

[6] John Franks. Change log for WN. WN distribution, `http://hopf.math.nwu.edu/`, August 1995.

[7] John Heidemann, Katia Obraczka, and Joe Touch. Modeling the performance of HTTP over several transport protocols. Submitted to IEEE/ACM Transactions on Networking, November 1996.

[8] Van Jacobson and Mike Karels. Congestion avoidance and control. *ACM Computer Communication Review*, 18(4):314–329, August 1990. Revised version of his SIGCOMM '88 paper.

[9] J.C. Mogul and S.E. Deering. Path MTU discovery. RFC 1191, Internet Request For Comments, November 1990.

[10] Jeffrey C. Mogul. The case for persistent-connection HTTP. In *Proceedings of the SIGCOMM '95*, pages 299–313. ACM, August 1995.

[11] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP latency. In *Proceedings of the Second International World Wide Web Conference*, October 1994.

[12] W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, 1994.

[13] W. Richard Stevens. *TCP/IP Illustrated*, volume 2. Addison-Wesley, 1995.

[14] Joe Touch. TCP control block interdependence. Work in progress (Internet draft draft-touch-tcp-interdep-00.txt, expires 11 December 1996), June 1996.

[15] Joe Touch, John Heidemann, and Katia Obraczka. Analysis of HTTP performance. Released as web page `http://www.isi.edu/lsam/publications-/http-perf/`, June 1996.