Network Performance Effects of HTTP/1.1, CSS1, and PNG

Henrik Frystyk Nielsen
World Wide Web Consortium
frystyk@w3.org

James Gettys
Visiting Scientist, World Wide Web Consortium
Digital Equipment Corporation
jg@pa.dec.com

Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, Chris Lilley
World Wide Web Consortium
abaird, eric, howcome, chris@w3.org

Abstract

We describe our investigation of the effect of persistent connections, pipelining and link level document compression on our client and server HTTP implementations. A simple test setup is used to verify HTTP/1.1's design and understand HTTP/1.1 implementation strategies. We present TCP and real time performance data between the libwww robot [27] and both the W3C's Jigsaw [28] and Apache [29] HTTP servers using HTTP/1.0, HTTP/1.1 with persistent connections, HTTP/1.1 with pipelined requests, and HTTP/1.1 with pipelined requests and deflate data compression [22]. We also investigate whether the TCP Nagle algorithm has an effect on HTTP/1.1 performance. While somewhat artificial and possibly overstating the benefits of HTTP/1.1, we believe the tests and results approximate some common behavior seen in browsers. The results confirm that HTTP/1.1 is meeting its major design goals. Our experience has been that implementation details are very important to achieve all of the benefits of HTTP/1.1.

For all our tests, a pipelined HTTP/1.1 implementation outperformed HTTP/1.0, even when the HTTP/1.0 implementation used multiple connections in parallel, under all network environments tested. The savings were at least a factor of two, and sometimes as much as a factor of ten, in terms of packets transmitted. Elapsed time improvement is less dramatic, and strongly depends on your network connection.

Some data is presented showing further savings possible by changes in Web content, specifically by the use of CSS style sheets [10], and the more compact PNG [20] image representation, both recent recommendations of W3C. Time did not allow full end to end data collection on these cases. The results show that HTTP/1.1 and changes in Web content will have dramatic results in Internet and Web performance as HTTP/1.1 and related technologies deploy over the near future. Universal use of style sheets, even without deployment of HTTP/1.1, would cause a very significant reduction in network traffic.

This paper does not investigate further performance and network savings enabled by the improved caching facilities provided by the HTTP/1.1 protocol, or by sophisticated use of range requests.

1 Introduction

Typical web pages today contain a HyperText Markup Language (HTML) document, and many embedded images. Twenty or more embedded images are quite common. Each of these images is an independent object in the Web, retrieved (or validated for change) separately. The common behavior for a web client, therefore, is to fetch the base HTML document, and then immediately fetch the embedded objects, which are typically located on the same server.

The large number of embedded objects represents a change from the environment in which the Web transfer protocol, the Hypertext Transfer Protocol (HTTP) was designed. As a result, HTTP/1.0 handles multiple requests from the same server inefficiently, creating a separate TCP connection for each object.

The recently released HTTP/1.1 standard was designed to address this problem by encouraging multiple transfers of objects over one connection. Coincidentally, expected changes in Web content are expected to decrease the number of embedded objects, which will improve network performance. The cheapest object is one that is no longer needed.

To test the effects of some of the new features of HTTP/1.1, we simulated two different types of client behavior: visiting a site for the first time, where nothing is in the client cache, and revalidating cached items when a site is revised. Tests were conducted in three different network environments designed to span a range of common web uses: a local Ethernet (LAN), transcontinental Internet (WAN), and a 28.8 Kbps dialup link using the Point-to-Point Protocol (PPP).

In this paper, we present the final results, and some of the thought processes that we went through while testing and optimizing our implementations. Our hope is that our experience may guide others through their own implementation efforts and help them avoid some non-obvious performance pits we fell into. Further information, the data itself (and later data collection runs) can be found on the Web [25].

1.1 Changes to HTTP

HTTP/1.1 [4] is an upward compatible protocol to HTTP/1.0 [3]. Both HTTP/1.0 and HTTP/1.1 use the TCP protocol [12] for data transport. However, the two versions of HTTP use TCP differently.

HTTP/1.0 opens and closes a new TCP connection for each operation. Since most Web objects are small, this practice means a high fraction of packets are simply TCP control packets used to open and close a connection. Furthermore, when a TCP connection is first opened, TCP employs an algorithm known as slow start [11]. Slow start uses the first several data packets to probe the network to determine the optimal transmission rate. Again, because Web objects are small, most objects are transferred before their TCP connection completes the slow start algorithm. In other words, most HTTP/1.0 operations use TCP at its least efficient. The results have been major problems due to resulting congestion and unnecessary overhead [6].

HTTP/1.1 leaves the TCP connection open between consecutive operations. This technique is called “persistent connections,” which both avoids the costs of multiple opens and closes and reduces the impact of slow start. Persistent connections are more efficient than the current practice of running multiple short TCP connections in parallel.

By leaving the TCP connection open between requests, many packets can be avoided, while avoiding multiple RTTs due to TCP slow start. The first few packet exchanges of a new TCP connection are either too fast, or too slow for that path. If these exchanges are too fast for the route (common in today’s Internet), they contribute to Internet congestion.

Conversely, since most connections are in slow start at any given time in HTTP/1.0 not using persistent connections, keeping a dialup PPP link busy has required running multiple TCP connections simultaneously (typical implementations have used 4 TCP connections). This can exacerbate the congestion problem further.

The “Keep-Alive” extension to HTTP/1.0 is a form of persistent connections. HTTP/1.1's design differs in minor details from Keep-Alive to overcome a problem discovered when Keep-Alive is used with more than one proxy between a client and a server.

Persistent connections allow multiple requests to be sent without waiting for a response; multiple requests and responses can be contained in a single TCP segment. This can be used to avoid many round trip delays, improving performance, and reducing the number of packets further. This technique is called “pipelining’ in HTTP.

HTTP/1.1 also enables transport compression of data types so those clients can retrieve HTML (or other) uncompressed documents using data compression; HTTP/1.0 does not have sufficient facilities for transport compression. Further work is continuing in this area [26].

The major HTTP/1.1 design goals therefore include:

- lower HTTP's load on the Internet for the same amount of “real work”, while solving the congestion caused by HTTP
- HTTP/1.0’s caching is primitive and error prone; HTTP/1.1 enable applications to work reliably with caching
- end user performance must improve, or it is unlikely that HTTP/1.1 will be deployed

HTTP/1.1 provides significant improvements to HTTP/1.0 to allow applications to work reliably in the face of caching, and to allow applications to mark more content cacheable. Today, caching is often deliberately defeated in order to achieve reliability. This paper does not explore these effects.

HTTP/1.1 does not attempt to solve some commonly seen problems, such as transient network overloads at popular web sites with topical news (e.g. the Schumacher-Levy comet impact on Jupiter), but should at least help these problems.

This paper presents measured results of the consequences of HTTP/1.1 transport protocol additions. Many of these additions have been available as extensions to HTTP/1.0, but this paper shows the possible synergy when the extensions to the HTTP protocol are used in concert, and in with changes in content.

1.2 Range Requests and Validation

To improve the perceived response time, a browser needs to learn basic size information of each object in a page (required for page layout) as soon as possible. The first bytes typically contain the image size. To achieve better concurrency and retrieve the first few bytes of embedded links while still receiving the bytes for the master document, HTTP/1.0 browsers usually use multiple TCP connections. We believe by using range requests HTTP/1.1 clients can achieve similar or better results over a single connection.

HTTP /1.1 defines as part of the standard (and most current HTTP/1.0 servers already implement) byte range facilities that allow a client to perform partial retrieval of objects. The initial intent of range requests was to allow caching proxy to finish interrupted transfers by requesting only the bytes of the document they currently do not hold in their cache.

To solve the problem that browsers need the size of embedded objects, we believe that the natural revalidation request for HTTP/1.1 will combine both cache validation headers and an *If-Range* request header, to prevent large objects from monopolizing the connection to the server over its connection. The range requested should be large enough to usually return any embedded metadata for the object for the common data types. This capability of HTTP/1.1 is implicit in its caching and range request design.

When a browser revisits a page, it has a very good idea what the type of any embedded object is likely to be, and can therefore both make a validation request and also simultaneously request the metadata of the embedded object if there has been any change. The metadata is much more valuable than the embedded image data. Subsequently, the browser might generate requests for the rest of the object, or for enough of each object to allow for progressive display of image data types (e.g. progressive PNG, GIF or JPEG images), or to multiplex between multiple large images on the page. We call this style of use of HTTP/1.1 "poor man's multiplexing."

We believe cache validation combined with range requests will likely become a very common idiom of HTTP/1.1.

1.3 Changes to Web Content

Roughly simultaneously to the deployment of the HTTP/1.1 protocol, (but not dependent upon it), the Web will see the deployment of Cascading Style Sheets (CSS) [30] and new image and animation formats such as Portable Network Graphics (PNG) [20] and Multiple-image Network Graphics (MNG) [31].

In the scientific environment where the Web was born, people were generally more concerned with the content of their documents than the presentation. In a research report, the choice of fonts matters less than the results being reported, so early versions of HyperText Markup Language (HTML) sufficed for most scientists. However, when non-scientific communities discovered the Web, the perceived limitations of HTML became a source of frustration. Web page designers with a background in paper-based desktop publishing wanted more control over the presentation of their documents than HTML was meant to provide. Cascading Style Sheets (CSS) offer many of the capabilities requested by page designers but is only now seeing widespread implementation.

In the absence of style sheets, authors have had to meet design challenges by twisting HTML out of shape, for instance, by studding their pages with small images that do little more than display text. In this section of the study, we estimate how Web performance will be affected by the introduction of CSS. We will not discuss other benefits to be reaped with style sheets, such as greater accessibility, improved printing, and easier site management.

On the web, most images are in GIF format. A new image format, PNG, has several advantages over GIF. PNG images render more quickly on the screen and - besides producing higher quality, cross-platform images - PNG images are usually smaller than GIF images.

MNG is an animation format in the PNG family, which - along with other advantages - is more compact than animated GIF.

2 Prior Work

Padmanabhan and Mogul [1] show results from a prototype implementation which extended HTTP to support both persistent connections and pipelining, and study latencies, throughput, and system overhead issues involved in persistent connections. This analysis formed the basic data and justification behind HTTP/1.1's persistent connection and pipelining design. HTTP/1.1 primarily relies on pipelining rather than introducing new HTTP methods to achieve the performance benefits documented below. As this paper makes clear, both pipelining and persistent connections are needed to achieve high performance over a single HTTP connection.

Pipelining, or batching, have been successfully used in a number of other systems, notably graphics protocols such as the X Window System [15] or Trestle [16], in its original RPC based implementation.

Touch, Heidemann, and Obraczka [5] explore a number of possible changes that might help HTTP behavior, including the sharing of TCP control blocks [19] and Transaction TCP (T/TCP) [17], [18]. The extended length of deployment of changes to TCP argued against any dependency of HTTP/1.1 on either of these; however, we believe that both mechanisms may improve performance, independently to the improvements made by HTTP/1.1. T/TCP might help reduce latency when revisiting a Web server after the server has closed its connection. Sharing of TCP control blocks would primarily help HTTP/1.0, however, since the HTTP/1.1 limits the number of connections between a client/server pair.

In independent work, Heidemann [7] describes the interactions of persistent connections with Nagle's algorithm. His experience is confirmed by our experience described in this paper, and by the experience of one of the authors with the X Window System, which caused the original introduction of the ability to disable Nagle's algorithm into BSD derived TCP implementations.

Simon Spero analyzed HTTP/1.0 performance [6] and prepared a proposal for a replacement for HTTP. HTTP/1.1, however, was constrained to maintain upward compatibility with HTTP/1.0. Many of his suggestions are worthwhile and should be explored further.

Style sheets have a long history in the Web [30]. We believe that the character of our results will likely be similar for other style sheet systems. However, we are not aware of any prior work investigating the network performance consequences of style sheets.

3 Test Setup

3.1 Test Web Site

We synthesized a test web site serving data by combining data (HTML and GIF image data) from two very heavily used home pages (Netscape and Microsoft) into one; hereafter called "*Microscape*". The initial layout of the Microscape web site was a single page containing typical HTML totaling 42KB with 42 inlined GIF images totaling 125KB. The embedded images range in size from 70B to 40KB; most are small, with 19 images less than 1KB, 7 images between 1KB and 2KB, and 6 images between 2KB and 3KB. While the resulting HTML page is larger, and contains more images than might be typical, such pages can be found on the Web.

3.2 First Time Retrieval Test

The first time retrieval test is equivalent to a browser visiting a site for the first time, e.g. its cache is empty and it has to retrieve the top page and all the embedded objects. In HTTP, this is equivalent to 43 *GET* requests.

3.3 Revalidate Test

This test is equivalent to revisiting a home page where the contents are already available in a local cache. The initial page and all embedded objects are validated, resulting in no actual transfer of the HTML or the embedded objects. In HTTP, this is equivalent to 43 *Conditional GET* requests. HTTP/1.1 supports two mechanisms for cache validation: *entity tags*, which are a guaranteed unique tag for a particular version of an object, and date stamps. HTTP/1.0 only supports the latter.

HTTP/1.0 support was provided by an old version of libwww (version 4.1D) which supported plain HTTP/1.0 with multiple simultaneous connections between two peers and no persistent cache. In this case we simulated the cache validation behavior by issuing HEAD requests on the images instead of *Conditional GET* requests. The profile of the HTTP/1.0 revalidation requests therefore was a total of 43 associated with the top page with one *GET* (HTML) and 42 *HEAD* requests (images), in the initial tests. The HTTP/1.1 implementation of libwww (version 5.1) differs from the HTTP/1.0 implementation. It uses a full HTTP/1.1 compliant persistent cache generating 43 *Conditional GET* requests with appropriate cache validation headers to make the test more similar to likely browser behavior. Therefore the number of packets in the results reported below for HTTP/1.0 are higher than of the correct cache validation data reported for HTTP/1.1.

3.4 Network Environments Tested

In order to measure the performance in commonly used different network environments, we used the following (Table 1) three combinations of bandwidth and latency:

Channel	Connection	RTT	MSS
High bandwidth, low latency	LAN - 10Mbit Ethernet	< 1ms	1460
High bandwidth, high latency	WAN – MA (MIT/LCS) to CA (LBL)	~ 90 ms	1460
Low bandwidth, high latency	PPP - 28.8k modem line using LCS dialup service	~ 150 ms	1460

Table 1 – Tested Network Environments

3.5 Applications, Machines and OSs

Several platforms were used in the initial stage of the experiments for running the HTTP servers. However, we ended up using relatively fast machines to try to prevent unforeseen bottlenecks in the servers and clients used. Jigsaw is written entirely in Java and relies on specific network features for controlling TCP provided only by Java Development Kit (JDK) 1.1. Apache is written in C and runs on multiple UNIX variants.

Component	Type and Version
Server Hardware	www26.w3.org, Sun SPARC Ultra-1, Solaris 2.5
LAN Client Hardware	zorch.w3.org , Digital AlphaStation 400 4/233, UNIX 4.0a
WAN Client Hardware	turn.ee.lbl.gov, Digital AlphaStation 3000, UNIX 4.0
PPP Client Hardware	big.w3.org , Dual Pentium Pro PC, Windows NT Server 4.0
HTTP Server Software	Jigsaw 1.06 and Apache 1.2b10
HTTP Client Software	libwww robot, Netscape Communicator 4.0 beta 5 and Microsoft Internet Explorer 4.0 beta 1 on Windows NT

Table 2 - Applications, Machines, and OSs

None of the machines were under significant load while the tests were run. The server is identical through our final tests - only the client changes connectivity and behavior. Both Jigsaw and Libwww are currently available with HTTP/1.1 implementations without support for the features described in this paper and Apache is in beta release. During the experiments changes were made to all three applications. These changes will be made available through normal release procedures for each of the applications.

4 Initial Investigations and Tuning

The HTTP/1.0 robot was set to use plain HTTP/1.0 requests using one TCP connection per request. We set the maximum number of simultaneous connections to 4, the same as Netscape Navigator's default (and hard wired maximum, it turns out).

After testing HTTP/1.0, we ran the robot as a simple HTTP/1.1 client using persistent connections. That is, the request / response sequence looks identical to HTTP/1.0 but all communication happens on the same TCP connection instead of 4, hence serializing all requests. The results as seen in Table 3 was a significant saving in TCP packets using HTTP/1.1 but also a big increase in elapsed time.

4.1 Pipelining

As a means to lower the elapsed time and improve the efficiency, we introduced pipelining into libwww. That is, instead of waiting on a response to arrive before issuing new requests, as many requests as possible are issued at once. The responses are still serialized and no changes were made to the HTTP messages; only the timing has changed as the robot has multiple outstanding requests on the same connection.

The robot generates quite small HTTP requests - our library implementation is very careful not to generate unnecessary headers and not to waste bytes on white space. The result is an average request size of around 190 bytes, which is significantly smaller than many existing product HTTP implementations, as seen in Table 10 and Table 11 below.

The requests are buffered before transmission so that multiple HTTP requests can be sent with the same TCP segment. This has a significant impact on the number of packets required to transmit the payload and lowers system time CPU usage by both client and server. However, this means that requests are not immediately transmitted, and we therefore need a mechanism to flush the output buffer. First we implemented a version with two mechanisms:

1. The buffer was flushed if the data in the output buffer reached a certain size. We experimented with the output buffer size and found that 1024 bytes is a good compromise. In case the MTU is 536 or 512 we will produce two full TCP segments, and if the MTU is 1460 (Ethernet size) then we can nicely fit into one segment.

2. We introduced a timer in the output buffer stream which would time-out after a specified period of time and force the buffer to be flushed. It is not clear what the optimal flush time-out period is but it is likely that it is a function of the network load and connectivity. Initially we used a 1 second delay for the initial results in Table 3, but used a 50 ms delay in for all later tests. Further work is required to understand where we should set such a timer, which might also take into account the RTT for this particular connection or other factors, to support old clients which do not explicitly flush the buffer.

	HTTP/1.0	HTTP/1.1 Persistent	HTTP/1.1 Pipeline
Max simultaneous sockets	6	1	1
Total number of sockets used	40	1	1
Packets from client to server	226	70	25
Packets from server to client	271	153	58
Total number of packets	497	223	83
Total elapsed time [secs]	1.85	4.13	3.02

Table 3 - Jigsaw - Initial High Bandwidth, Low Latency Cache Revalidation Test

We were simultaneously very happy and quite disappointed with the initial results above, taken late at night on a quiet Ethernet. Elapsed time performance of HTTP/1.1 with pipelining was worse than HTTP/1.0 in this initial implementation, though the number of packets used were dramatically better. We scratched our heads for a day, then convinced ourselves that on a local Ethernet, there was no reason that HTTP/1.1 should ever perform more slowly than HTTP/1.0. The local Ethernet cannot suffer from fairness problems that might give multiple connections a performance edge in a long haul network. We dug into our implementation further.

4.1.1 Buffer Tuning

After study, we realized that the application (the robot) has much more knowledge about the requests than libwww, and by introducing an explicit flush mechanism in the application, we could get significantly better performance. We modified the robot to force a flush after issuing the first request on the HTML document and then buffer the following requests on the inlined images. While HTTP libraries can be arranged to automatically flush buffers automatically after a timeout, taking advantage of knowledge in the application can result in a considerably faster implementation than relying on such a timeout.

4.1.2 Nagle Interaction

We expected, due to experience of one of the authors, that a pipelined implementation of HTTP might encounter the Nagle algorithm [2] [5] in TCP. The Nagle algorithm was introduced in TCP as a means of reducing the number of small TCP segments by delaying their transmission in hopes of further data becoming available, as commonly occurs in *telnet* or *rlogin* traffic. As our implementation can generate data asynchronously without waiting for a response, the Nagle algorithm could be a bottleneck.

A pipelined application implementation buffers its output before writing it to the underlying TCP stack, roughly equivalent to what the Nagle algorithm does for telnet connections. These two buffering algorithms tend to interfere, and using them together will often cause very significant performance degradation. For each connection, the server maintains a response buffer that it flushes either when full, or when there is no more requests coming in on that connection, or before it goes idle. This buffering enables aggregating responses (for example, cache validation responses) into fewer packets even on a high-speed network, and saving CPU time for the server.

In order to test this, we turned the Nagle algorithm off in both the client and the server. This was the first change to the server - all other changes were made in the client. In our initial tests, we did not observe significant problems introduced by Nagle's algorithm, though with hindsight, this was the result of our pipelined implementation and the specific test cases chosen, since with effective buffering, the segment sizes are large, avoiding Nagle's algorithm. In later experiments in which the buffering behavior of the implementations were changed, we did observe significant (sometimes dramatic) transmission delays due to Nagle; we recommend therefore that HTTP/1.1 implementations that buffer output disable Nagle's algorithm (set the `TCP_NODELAY` socket option). This confirms the experiences of Heidemann [7].

We also performed some tests against the Apache 1.2b2 server, which also supports HTTP/1.1, and observed essentially similar results to Jigsaw. Its output buffering in that initial beta test release was not yet as good as our revised version of Jigsaw, and in that release it processes at most five requests before terminating a TCP connection. When using pipelining, the number of HTTP requests served is often a poor indicator for when to close the connection. We discussed these results with Dean Gaudet and others of the Apache group and similar changes were made to the Apache server; our final results below are using a version of Apache 1.2b10.

4.2 Connection Management

Implementations need to close connections carefully. HTTP/1.0 implementations were able to naively close both halves of the TCP connection simultaneously when finishing the processing of a request. A pipelined HTTP/1.1 implementation can cause major problems if it does so.

The scenario is as follows: An HTTP server can close its connection between any two responses. An HTTP/1.1 client talking to a HTTP/1.1 server starts pipelining a batch of requests, for example 15 requests, on an open TCP connection. The server might decide that it will not serve more than 5 requests per connection and closes the TCP connection in both directions after it successfully has served the first five requests. The remaining 10 requests that are already sent from the client will along with client generated TCP ACK packets arrive on a closed port on the server. This "extra" data causes the server's TCP to issue a reset; this forces the client TCP stack to pass the last ACK'ed packet to the client application and discard all other packets. This means that HTTP responses that are either being received or already have been received successfully but haven't been ACK'ed will be dropped by the client TCP. In this situation the client does not have any means of finding out which HTTP messages were successful or even why the server closed the connection. The server may have generated a "*Connection: Close*" header in the 5th response but the header may have been lost due to the TCP reset, if the server's sending side is closed before the receiving side of the connection. Servers must therefore close each half of the connection independently.

TCP's congestion control algorithms [11] work best when there are enough packets in a connection that TCP can determine the approximate optimal maximum rate at which to insert packets into the Internet. Observed packet trains in the Internet have been dropping [13], almost certainly due to HTTP/1.0's behavior, as demonstrated in the data above, where a single connection rarely involves more than 10 packets, including TCP open and close. Some IP switch technology exploits packet trains to enable faster IP routing. In the tests above, the packet trains are significantly longer, but not as long as one might first expect, since fewer, larger packets are transmitted due to pipelining.

The HTTP/1.1 proposed standard specification does specify at most two connections to be established between a client/server pair. (If you get a long, dynamically generated document, a second connection might be required to fetch embedded objects.) Dividing the mean length of packet trains down by a factor of two diminish the benefits to the Internet (and possibly to the end user due to slow start) substantially. Range requests need to be exploited to enable good interactive feel in Web browsers while using a single connection. Connections should be maintained as long as makes reasonable engineering sense [9], to pick up user's "click ahead" while following links.

5 After Initial Tuning Tests

To make our final round of tests as close as possible to likely real implementations, we took the opportunity to change the HTTP/1.1 version of the robot to issue full HTTP/1.1 cache validation requests. These use *If-None-Match* headers and opaque validators, rather than the *HEAD* requests used in our HTTP/1.0 version of the robot. With the optimized clients and servers, we then took a complete set of data, for both the first time retrieval and cache validation tests, in the three network environments.

It was easiest to implement full HTTP/1.1 caching semantics by enabling persistent caching in libwww. This had unexpected consequences due to libwww's implementation of persistent caching, which is written for ease of porting and implementation rather than performance. Each cached object contains two independent files: one containing the cacheable message headers and the other containing the message body. This would be an area that one would optimize carefully in a product implementation; the overhead in our implementation became a performance bottleneck in our HTTP/1.1 tests. Time and resources did not permit optimizing this code. Our final measurements use correct HTTP/1.1 cache validation requests, and run with a persistent cache on a memory file system to reduce the disk performance problems that we observed.

The measurements in Table 4 through Table 9 are a consistent set of data taken just before publication. While Jigsaw had outperformed Apache in the first round of tests, Apache now outperforms Jigsaw (which ran interpreted in our tests). Results of runs generally resembled each other. For the WAN test however, the higher the latency, the better HTTP/1.1 performed. The data below was taken when the Internet was particularly quiet.

5.1 Changing Web Content Representation

After having determined that HTTP/1.1 outperforms HTTP/1.0 we decided to try other means of optimizing the performance. We therefore investigated how much we would gain by using data compression of the HTTP message body. That is, we do not compress the HTTP headers, but only the body using the "*Content-Encoding*" header to describe the encoding mechanism. We use the zlib compression library [23] version 1.04, which is a freely available C based code base. It has a stream based interface which interacts nicely with the libwww stream model. Note that the PNG library also uses zlib, so common implementations will share the same data compression code. Implementation was at most a day or two.

The client indicates that it is capable of handling the "*deflate*" content coding by sending an "*Accept-Encoding: deflate*" header in the requests. In our test, the server does not perform on-the-fly compression but sends out a pre-computed deflated version of the Microscape HTML page. The client performs on-the-fly inflation and parses the inflated HTML using its normal HTML parser.

Note that we only compress the HTML page (the first GET request) and *not* any of the following images, which are already compressed using various other compression algorithms (GIF).

The zlib library has several flags for how to optimize the compression algorithm, however we used the default values for both deflating and inflating. In our case this caused the Microscape HTML page to be compressed more than a factor of three from 42K to 11K. This is a typical factor of gain using this algorithm on HTML files. This means that we decrease the overall payload with about 31K or approximately 19%.

6 Measurements

The datga shown in these tables are a summary of the more detailed data acquisition overview. In all cases, the traces were taken on client side, as this is where the interesting delays are. Each run was repeated 5 times in order to make up for network fluctuations, except Table 10 and Table 11, which were repeated three times. In the tables below, Pa = Packets, and Sec = Seconds. %ov is the percentage of overhead bytes due to TCP/IP packet headers.

	First Time Retrieval				Cache Validation			
	Pa	Bytes	Sec	%ov	Pa	Bytes	Sec	%ov
HTTP/1.0	510.2	216289	0.97	8.6	374.8	61117	0.78	19.7
HTTP/1.1	281.0	191843	1.25	5.5	133.4	17694	0.89	23.2
HTTP/1.1 Pipelined	181.8	191551	0.68	3.7	32.8	17694	0.54	6.9
HTTP/1.1 Pipelined w. compression	148.8	159654	0.71	3.6	32.6	17687	0.54	6.9

Table 4 - Jigsaw - High Bandwidth, Low Latency

	First Time Retrieval				Cache Validation			
	Pa	Bytes	Sec	%ov	Pa	Bytes	Sec	%ov
HTTP/1.0	489.4	215536	0.72	8.3	365.4	60605	0.41	19.4
HTTP/1.1	244.2	189023	0.81	4.9	98.4	14009	0.40	21.9
HTTP/1.1 Pipelined	175.8	189607	0.49	3.6	29.2	14009	0.23	7.7
HTTP/1.1 Pipelined w. compression	139.8	156834	0.41	3.4	28.4	14002	0.23	7.5

Table 5 - Apache - High Bandwidth, Low Latency

	First Time Retrieval				Cache Validation			
	Pa	Bytes	Sec	%ov	Pa	Bytes	Sec	%ov
HTTP/1.0	565.8	251913	4.17	8.2	389.2	62348.0	2.96	20.0
HTTP/1.1	304.0	193595	6.64	5.9	137.0	18065.6	4.95	23.3
HTTP/1.1 Pipelined	214.2	193887	2.33	4.2	34.8	18233.2	1.10	7.1
HTTP/1.1 Pipelined w. compression	183.2	161698	2.09	4.3	35.4	19102.2	1.15	6.9

Table 6 - Jigsaw - High Bandwidth, High Latency

	First Time Retrieval				Cache Validation			
	Pa	Bytes	Sec	%ov	Pa	Bytes	Sec	%ov
HTTP/1.0	559.6	248655.2	4.09	8.3	370.0	61887	2.64	19.3
HTTP/1.1	309.4	191436.0	6.14	6.1	104.2	14255	4.43	22.6
HTTP/1.1 Pipelined	221.4	191180.6	2.23	4.4	29.8	15352	0.86	7.2
HTTP/1.1 Pipelined w. compression	182.0	159170.0	2.11	4.4	29.0	15088	0.83	7.2

Table 7 - Apache - High Bandwidth, High Latency

	First Time Retrieval				Cache Validation			
	Pa	Bytes	Sec	%ov	Pa	Bytes	Sec	%ov
HTTP/1.1	309.6	190687	63.8	6.1	89.2	17528	12.9	16.9
HTTP/1.1 Pipelined	284.4	190735	53.3	5.6	31.0	17598	5.4	6.6
HTTP/1.1 Pipelined w. compression	234.2	159449	47.4	5.5	31.0	17591	5.4	6.6

Table 8 - Jigsaw - Low Bandwidth, High Latency

	First Time Retrieval				Cache Validation			
	Pa	Bytes	Sec	%ov	Pa	Bytes	Sec	%ov
HTTP/1.1	308.6	187869	65.6	6.2	89.0	13843	11.1	20.5
HTTP/1.1 Pipelined	281.4	187918	53.4	5.7	26.0	13912	3.4	7.0
HTTP/1.1 Pipelined w. compression	233.0	157214	47.2	5.6	26.0	13905	3.4	7.0

Table 9 - Apache - Low Bandwidth, High Latency

	First Time Retrieval				Cache Validation			
	Pa	Bytes	Sec	%ov	Pa	Bytes	Sec	%ov
Netscape Navigator[1]	339.4	201807	58.8	6.3	108	19282	14.9	18.3
Internet Explorer[2]	360.3	199934	63.0	6.7	301.0	61009	17.0	16.5

Table 10 – Jigsaw - Netscape Navigator and MS Internet Explorer, Low Bandwidth, High Latency

	First Time Retrieval				Cache Validation			
	Pa	Bytes	Sec	%ov	Pa	Bytes	Sec	%ov
Netscape Netscape[1]	334.3	199243	58.7	6.3	103.3	23741	5.9	14.8
Internet Explorer[2]	381.3	204219	60.6	6.9	117.0	23056	8.3	16.9

Table 11 – Apache - Netscape Navigator and MS Internet Explorer, Low Bandwidth, High Latency

7 Observations on HTTP/1.0 and 1.1 Data

Buffering requests and responses significantly reduces the number of packets required. For a common operation in the Web (revisiting a page cached locally), our HTTP/1.1 with buffered pipelining implementation uses less than 1/10 of the total number of packets that HTTP/1.0 does, and executes in much less elapsed time, using a single TCP connection on a WAN. This is a factor of three improvement over HTTP/1.1 implemented without buffering of requests and responses. HTTP requests are usually highly redundant and the actual number of bytes that changes between requests can be as small as 10%. Therefore, a more compact wire representation for HTTP could increase pipelining's benefit for cache revalidation further up to an additional factor of five or ten, from back of the envelope calculations based on the number of bytes changing from one request to the next.

An HTTP/1.1 implementation that does not implement pipelining will perform worse (have higher elapsed time) than an HTTP/1.0 implementation using multiple connections.

The mean number of packets in a TCP session increased between a factor of two and a factor of ten. The mean size of a packet in our traffic roughly doubled. However, if style sheets see widespread use, do not expect as large an improvement in the number of packets in a TCP session, as style sheets may eliminate unneeded image transfers, shortening packet trains.

Since fewer TCP segments were significantly bigger and could almost always fill a complete Ethernet segment, server performance also increases when using pipelined requests, even though only the client changed behavior.

For the first time retrieval test, bandwidth savings due to pipelining and persistent connections of HTTP/1.1 is only a few percent. Elapsed time on both WAN and LAN roughly halved.

8 Compression Issues

8.1 Why Compression is Important

The first few packets of a new TCP connections are controlled by the TCP slow start algorithm. The first TCP packet of payload (that is, not part of the TCP open handshake) on an Ethernet contains about 1400 bytes (the HTTP response header and the first part of the HTML body). When this first packet arrives, the client starts parsing the HTML in order to present the contents to the end-user. Any inlined objects referenced in the first segment are very likely to be on the same server and therefore can be pipelined onto the same TCP connection. If the number of new requests generated by the parser exceeds the pipeline output buffer size then a new batch of HTTP requests can be sent immediately - otherwise in our implementation, the batch is delayed until either of the pipeline flush mechanisms is triggered.

If the next batch of requests is delayed, then no new data is written to the client TCP stack. Since the connection is in slow start, the server can not send any more data until it gets an ACK from the first segment. TCP can either piggy back its ACK onto an outgoing packet or generate a separate ACK packet. In either case, the end result is an extra delay causing overall performance degradation on a LAN.

A separate ACK packet is subject to the delayed acknowledgement algorithm that may delay the packet up to 200ms. Another strategy, which we have not tried, would be to always flush the buffer after processing the first segment if no new segments are available, though this would often cost an extra packet.

We observed these delayed ACKs in out traces on the first packet sent from the server to the client. In the *Pipelining* case, the HTML text (sent in clear) did not contain enough information to force a new batch of requests. In the *Pipelining and HTML compression* case, the first packet contains approximately 3 times as much HTML so the probability of having enough requests to immediately send a new batch is higher.

HTML compression is therefore very important since it increases the probability that there are enough inlined objects in the first segment to immediately issue a new batch without introducing any extra delay (or conversely, generating more packets than otherwise necessary.)

This indicates that the relationship between payload, TCP packets and transfer time is non-linear and that the first packets on a connections are relatively more "expensive" than later packets.

The exact results may depend on how the slow start algorithm is implemented on the particular platform. Some TCP stacks implement slow start using one TCP segment whereas others implement it using two packets

In our tests, the client is always decompressing compressed data

---

[1] These measurements were performed using max 4 (default) simultaneous connections and HTTP/1.0 Keep-Alive headers. The Netscape HTTP client implementation uses the HTTP/1.0 Keep-Alive mechanism to allow for multiple HTTP messages to be transmitted on the same TCP connection. It therefore used 8 connections compared to 42 for the libwww HTTP/1.0 implementation, in which this feature was disabled.

[2] The measurements were performed using max 6 (default) simultaneous connections and HTTP/1.0 Keep-Alive headers. As with Netscape Communicator, it uses the HTTP/1.0 Keep-Alive mechanism to allow for multiple HTTP messages to be transmitted on the same TCP connection. The total number of connections used in the test case is 6.

on the fly. This test does not take into account the time it would take to compress an HTML object on the fly and whether this will take longer than the time gained transmitting fewer packets. Further experiments are needed to see if compression of dynamically generated content would save CPU time over transferring the data. Static content can be compressed in advance and may not take additional resources on the server

8.2 Summary of Compression Performance

Transport compression helped in all environments and enabled significant savings (about 16% of the packets and 12% of the elapsed time in our first time retrieval test); the decompression time for the client is more than offset by the savings in data transmission. Deflate compression is more efficient than the data compression algorithms used in modems (see section 8.3). Your mileage will vary depending on precise details of your Internet connection.

For clients that do not load images, transport compression should provide a major gain. Faster retrieval of HTML pages will also help time to render significantly, for all environments.

8.3 Further Compression Experiments

We also performed a simple test confirming that zlib compression is significantly better than the data compression found in current modems [24]. The compression used the zlib compression algorithm and the test is done on the HTML page of the Microscape test site. We performed the HTML retrieval (a single HTTP GET request) only with no embedded objects. The test was run over standard 28.8Kbps modems.

	Jigsaw		Apache	
	Pa	Sec	Pa	Sec
Uncompressed HTML	67	12.21	67	12.13
Compressed HTML	21.0	4.35	4.35	4.43
Saved using compression	68.7%	64.4%	68.7%	64.5%

The default compression provided by zlib gave results very similar to requesting best possible compression to minimize size.

Case of HTML tags can effect compression. Compression is significantly worse (.35 rather than .27) if mixed case HTML tags are used. The best compression was found if all HTML tags were uniformly lower case, (since the compression dictionary can reuse what are common English words). HTML tool writers should beware of this result, and we recommend HTML tags be uniformly lower case for best performance of compressed documents.

9 Impact of Changing Web Content

In the preceding section, the compression experiments did not take advantage of knowledge of the content that was transmitted. By examining the content (text and images) of documents, they can be re-expressed in more compact and powerful formats while retaining visual fidelity.

This section explores how CSS, PNG and MNG may be used to compression content. We converted the images in our test page to PNG, animations to MNG, and where possible replaced images with HTML and CSS.

9.1 Replacing Images with HTML and CSS

While CSS give page designers and readers greater control of page presentation, it has the added value of speeding up page downloads. First of all, modularity in style sheets means that the same style sheet may apply to many documents, thus reducing the need to send redundant presentation information over the network.

Second, CSS can eliminate small images used to represent symbols (such as bullets, arrows, spacers, etc.) that appear in fonts for the Unicode character set. Replacing images with CSS reduces the number of separate resources referenced, and therefore reduces the protocol requests and possible name resolutions required to retrieve them.

Third, CSS gives designers greater control over the layout of page elements, which will eliminate the practice of using invisible images for layout purposes. Images may now be images -- be seen and not waited for.

The Microscape test page contains 40 static GIF images, many of which may be replaced by HTML+CSS equivalents. Figure 1 shows one such image that requires 682 bytes.

Figure 1 – "solutions" GIF

The image depicts a word ("solutions") using a certain font (a bold, oblique sans-serif, approximately 20 pixels high) and color combination (white on a yellowish background) and surrounding it with some space. Using HTML+CSS, the same content can be represented with the following phrase:

```
P.banner {
    color: white;
    background: #FC0;
    font: bold oblique 20px sans-serif;   CSS
    padding: 0.2em 10em 0.2em 1em;
  }

<P CLASS=banner>solutions                 HTML
```

The HTML and CSS version only takes up around 150 bytes. When displayed in a browser that supports CSS, the output is similar to the image. Differences may occur due to unavailability of fonts, of anti-aliasing and the use of non-pixel units.

Replacing this image with HTML and CSS has two implications for performance. First, the number of bytes needed to represent the content is reduced by a factor of more than 4, even before any transport compression is applied. Second, there is no need to fetch the external image, and since HTML and CSS can coexist in the same file one HTTP request is saved.

Trying to replicate all 40 images on the Microscape test page reveals that:

- 22 of the 40 images can be represented in HTML+CSS. Encoded in GIF, these images take up 14791 bytes, and the HTML+CSS replacement is approximately 3200 bytes, a savings factor of around 4.6. This factor will increase further if compression is applied to the HTML+CSS code.

- Further, 3 images can be reduced to roughly half their size by converting part of their content to HTML+CSS. Their current size is 7541 bytes, and the HTML+CSS demo-replacement is 610 bytes.

- The elimination of 22 HTTP requests would save approximately 4600 bytes transmitted and the approximately 4300 bytes received, presuming the length of the requests (210 bytes) and responses (192 bytes for cache validation, and ~240 bytes for an actual successful GET request). This slightly overstates the savings; many style sheets will be stored separate from the documents and cached independently.

- 14 of the 40 images, taking up 80601 bytes, cannot be represented in HTML+CSS1. These are photographs, non-textual graphics, or textual effects beyond CSS (e.g. rotated text). However, these images can be converted to PNG.

It should be noted that the HTML+CSS sizes are estimates based on a limited test set, but the results indicate that style sheets may make a very significant impact on bandwidth (and end user delays) of the web. At the time of writing, no CSS browser can render all the replacements correctly.

9.2 Converting images from GIF to PNG and MNG

The 40 static GIF images on the test page totaled 103,299 bytes, much larger than the size of the HTML file. Converting these images to PNG using a standard batch process (giftopnm, pnmtopng) resulted in a total of 92,096 bytes, saving 11,203 bytes.

The savings are modest because many of the images are very small. PNG does not perform as well on the very low bit depth images in the sub-200 byte category because its checksums and other information make the file a bit bigger even though the actual image data is often smaller.

The two GIF animations totaled 24,988 bytes. Conversion to MNG gave a total of 16,329 bytes, a saving of 8,659 bytes.

It is clear that this sample is too small to draw any conclusions on typical savings (~19% of the image bytes, or ~10% of the total payload bandwidth, in this sample) due to PNG and MNG. Note that the converted PNG and MNG files contain gamma information, so that they display the same on all platforms; this adds 16 bytes per image. GIF images do not contain this information.

A very few images in our data set accounted for much of the total size. Over half of the data was contained in a single image and two animations. Care in selection of images is clearly very important to good design.

10 Implementation Experience

Pipelining implementation details can make a very significant difference on network traffic, and bear some careful thought, understanding, and testing. To take full advantage of pipelining, applications need explicit interfaces to flush buffers and other minor changes.

The read buffering of an implementation, and the details of how urgently data is read from the operating system, can be very significant to get optimal performance over a single connection using HTTP/1.1. If too much data accumulates in a socket buffer TCP may delay ACKs by 200ms. Opening multiple connections in HTTP/1.0 resulted in more socket buffers in the operating system, which as a result imposed lower requirements of speed on the application, while keeping the network busy.

We estimate two people for two months implemented the work reported on here, starting from working HTTP/1.0 implementations. We expect others leveraging from the experience reported here might accomplish the same result in much less time, though of course we may be more expert than many due to our involvement in HTTP/1.1 design.

10.1 Tools

Our principle data gathering tool is the widely available *tcpdump* program [14]; on Windows we used Microsoft's NetMon program. We also used Tim Shepard's *xplot* program [8] to graphically plot the dumps; this was very useful to find a number of problems in our implementation not visible in the raw dumps. We looked at data in both directions of the TCP connections. In the detailed data summary, there are direct links to all dumps in xplot formats. The *tcpshow* program [21] was very useful when we needed to see the contents of packets to understand what was happening.

11 Future Work

We believe the CPU time savings of HTTP/1.1 is very substantial due to the great reduction in TCP open and close and savings in packet overhead, and could now be quantified for Apache (currently the most popular Web server on the Internet). HTTP/1.1 will increase the importance of reducing parsing and data transport overhead of the very verbose HTTP protocol, which, for many operations, has been swamped by the TCP open and close overhead required by HTTP/1.0. Optimal server implementations for HTTP/1.1 will likely be significantly different than current servers.

Connection management is worth further experimentation and modeling. Padmanabhan [1] gives some guidance on how long connections should be kept open, but this work needs updating to reflect current content and usage of the Web, which have changed significantly since completion of the work.

Persistent connections, pipelining, transport compression, as well as the widespread adoption of style sheets (e.g. CSS) and more compact image representations (e.g. PNG) will increase the relative overhead of the very verbose HTTP text based protocol. These are most critical for high latency and low bandwidth environments such as cellular telephones and other wireless devices. A binary encoding or tokenized compression of HTTP and/or a replacement for HTTP will become more urgent given these changes in the infrastructure of the Web.

We have not investigated perceived time to render (our browser has not yet been optimized to use HTTP/1.1 features), but with the range request techniques outlined in this paper, we believe HTTP/1.1 can perform well over a single connection. PNG also provides time to render benefits relative to GIF. The best strategies to optimize time to render are clearly significantly different from those used by HTTP/1.1.

Serious analysis of trace data is required to quantify actual expected bandwidth gains from transport compression. At best, the results here can motivate such research.

Future work worth investigating includes other compression algorithms and the use of compression dictionaries optimized for HTML and CSS1 text.

12 Conclusions

For HTTP/1.1 to outperform HTTP/1.0 in elapsed time, an implementation must implement pipelining. Properly buffered pipelined implementations will gain additional performance and reduce network traffic further.

HTTP/1.1 implemented with pipelining outperformed HTTP/1.0, even when the HTTP/1.0 implementation uses multiple connections in parallel, under all circumstances tested. In terms of packets transmitted, the savings are typically at least a factor of two, and often much more, for our tests. Elapsed time improvement is less dramatic, but significant.

We experienced a series of TCP implementation bugs on our server platform (Sun Solaris) which could only be detected by examining the TCP dumps carefully. We believe the data in this paper to be unaffected by these problems. Implementers of HTTP doing performance studies should not presume that their platform's TCP implementation is bug free, and must be prepared to examine TCP dumps carefully both for HTTP and TCP implementation bugs to maximize performance.

The savings in terms of number of packets of HTTP/1.1 are truly dramatic. Bandwidth savings due to HTTP/1.1 and associated techniques are more modest (between 2% and 40% depending on the techniques used). Therefore, the HTTP/1.1 work on caching is as important as the improvements reported in this paper to save total bandwidth on the Internet. Network overloads caused by information of topical interest also strongly argue for good caching systems. A back of the envelope calculation shows that if all techniques described in this paper were applied, our test page might be downloaded over a modem in approximately 60% of the time of HTTP/1.0 browsers without significant change to the visual appearance. The addition of transport compression in HTTP/1.1 provided the largest bandwidth savings, followed by style sheets, and finally image format conversion, for our test page.

We believe HTTP/1.1 will significantly change the character of traffic on the Internet (given HTTP's dominant fraction of Internet traffic). It will result in significantly larger mean packet sizes, more packets per TCP connection, and drastically fewer packets contributing to congestion (by elimination of most packets due to TCP open and close, and packets transmitted before the congestion state of the network is known).

Due to pipelining HTTP/1.1 changes dramatically the "cost" and performance of HTTP, particularly for revalidating cached items. As a result, we expect that applications will significantly change their behavior. For example, caching proxies intended to enable disconnected operation may find it feasible to perform much more extensive cache validation than was feasible with HTTP/1.0. Researchers and product developers should be very careful when extrapolating from current Internet and HTTP server log data future web or Internet traffic and should plan to rework any simulations as these improvements to web infrastructure deploy.

Changes in web content enabled by deployment of style sheets; more compact image, graphics and animation representations will also significantly improve network and perceived performance during the period that HTTP/1.1 is being deployed. To our surprise, style sheets promise to be the biggest possibility of major network bandwidth improvements, whether deployed with HTTP/1.0 or HTTP/1.1, by significantly reducing the need for inlined images to provide graphic elements, and the resulting network traffic. Use of style sheets whenever possible will result in the greatest observed improvements in downloading new web pages, without sacrificing sophisticated graphics design.

13 References

[1] Padmanabhan, V. N. and J. Mogul, "Improving HTTP Latency," *Computer Networks and ISDN Systems*, v.28, pp. 25-35, Dec. 1995. Slightly Revised Version in *Proceedings of the 2nd International WWW Conference '94: Mosaic and the Web*, Oct. 1994.

[2] Nagle, J., "Congestion Control in IP/TCP Internetworks," RFC 896, Ford Aerospace and Communications Corporation, January 1984.

[3] Berners-Lee, Tim, R. Fielding, H. Frystyk. "Informational RFC 1945 - Hypertext Transfer Protocol -- HTTP/1.0," MIT/LCS, UC Irvine, May 1996.

[4] Fielding, R., J. Gettys, J.C. Mogul, H. Frystyk, T. Berners-Lee, "RFC 2068 - Hypertext Transfer Protocol -- HTTP/1.1," UC Irvine, Digital Equipment Corporation, MIT.

[5] Touch, J., J. Heidemann, K. Obraczka, "Analysis of HTTP Performance," USC/Information Sciences Institute, June, 1996.

[6] Spero, S., "Analysis of HTTP Performance Problems," http://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html July 1994.

[7] Heidemann, J., "Performance Interactions Between P-HTTP and TCP Implementation," *ACM Computer Communication Review*, 27 2, 65-73, April 1997.

[8] Shepard, T., Source for this very useful program is available at ftp://mercury.lcs.mit.edu/pub/shep. S.M. thesis "TCP Packet Trace Analysis." The thesis can be ordered from MIT/LCS Publications. Ordering information can be obtained from +1 617 253 5851 or send mail to publications@lcs.mit.edu. Ask for MIT/LCS/TR-494.

[9] Mogul, J. "The Case for Persistent-Connection HTTP", *Western Research Laboratory Research Report 95/4*, http://www.research.digital.com/wrl/publications/abstracts/95.4.html, Digital Equipment Corporation, May 1995.

[10] Lie, H., B. Bos, "Cascading Style Sheets, level 1," W3C Recommendation, World Wide Web Consortium, 17 Dec 1996.

[11] Jacobson, Van, "Congestion Avoidance and Control." *Proceedings of ACM SIGCOMM '88,* page 314-329. Stanford, CA, August 1988.

[12] Postel, Jon B., "Transmission Control Protocol," RFC 793, Network Information Center, SRI International, September 1981.

[13] Paxson, V., "Growth Trends in Wide-Area TCP Connections," *IEEE Network*, Vol. 8 No. 4, pp. 8-17, July 1994.

[14] Jacobson, V., C. Leres, and S. McCanne, *tcpdump*, available at ftp://ftp.ee.lbl.gov/tcpdump.tar.Z.

[15] Scheifler, R.W., J. Gettys, "The X Window System," *ACM Transactions on Graphics # 63*, Special Issue on User Interface Software.

[16] Manasse, Mark S., and Greg Nelson, "Trestle Reference Manual," *Digital Systems Research Center Research Report # 68*, December 1991.

[17] Braden, R., "Extending TCP for Transactions -- Concepts," RFC-1379, USC/ISI, November 1992.

[18] Braden, R., "T/TCP -- TCP Extensions for Transactions: Functional Specification," RFC-1644, USC/ISI, July 1994.

[19] Touch, J., "TCP Control Block Interdependence," RFC 2140, USC/ISI, April 1997.

[20] Boutell, T., T. Lane et. al. "PNG (Portable Network Graphics) Specification," W3C Recommendation, October 1996, RFC 2083, Boutell.Com Inc., January 1997. http://www.w3.org/pub/WWW/Graphics/PNG has extensive PNG information.

[21] Ryan, M., *tcpshow*, I.T. NetworX Ltd., 67 Merrion Square, Dublin 2, Ireland, June 1996.

[22] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951, Aladdin Enterprises, May 1996.

[23] Deutsch, L. Peter, Jean-Loup Gailly, "ZLIB Compressed Data Format Specification version 3.3," RFC 1950, Aladdin Enterprises, Info-ZIP, May 1996.

[24] "Recommendation V.42bis (01/90 – Data Compression procedures for data circuit terminating equipment (DCE) using error correction procedures," ITU, Geneva, Switzerland, January 1990.

[25] Online summary of results and complete data can be found at http://www.w3.org/Protocols/HTTP/Performance/.

[26] Mogul, Jeffery, Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, "Potential benefits of delta-encoding and data compression for HTTP," *Proceedings of ACM SIGCOMM '97*, Cannes France, September 1997.

[27] Nielsen, Henrik Frystyk, "Libwww - the W3C Sample Code Library," World Wide Web Consortium, April 1997. Source code is available at http://www.w3.org/Library.

[28] Baird-Smith, Anselm, "Jigsaw: An object oriented server," World Wide Web Consortium, February 1997. Source and other information are available at http://www.w3.org/Jigsaw.

[29] The Apache Group, "The Apache Web Server Project." The Apache Web server is the most common Web server on the Internet at the time of this paper's publication. Full source is available at http://www.apache.org.

[30] A Web page pointing to style sheet information in general can be found at http://www.w3.org/Style/.

[31] Multiple-image Network Graphics Format (MNG), version 19970427. ftp://swrinde.nde.swri.edu/pub/mng/documents/draft-mng-19970427.html.

14 Acknowledgements

Jeff Mogul of Digital's Western Research Laboratory has been instrumental in making the case for both persistent connections and pipelining in HTTP. We are very happy to be able to produce data with a real implementation confirming his and V.N. Padmanabhan's results and for his discussions with us about several implementation strategies to try.

Our thanks to Sally Floyd, Van Jacobson, and Craig Leres for use of a machine at Lawrence Berkeley Labs for the high bandwidth/high latency test.

Our thanks to Dean Gaudet of the Apache group for his timely cooperation to optimize Apache's HTTP/1.1 implementation.

Ian Jacobs wrote a more approachable summary of this work; some of it was incorporated into our introduction.

Our thanks to John Heidemann of ISI for pointing out one Solaris TCP performance problem we had missed in our traces. Our thanks to Jerry Chu of Sun Microsystems for his help working around the TCP problems we uncovered.

Digital Equipment Corporation supported Jim Gettys' participation.

The World Wide Web Consortium supported this work.