Improving HTTP Latency

Venkata N. Padmanabhan (University of California -- Berkeley)

Jeffrey C. Mogul (Digital Equipment Corporation Western Research Laboratory)

Abstract

The HTTP protocol, as currently used in the World Wide Web, uses a separate TCP connection for each file requested. This adds significant and unnecessary overhead, especially in the number of network round trips required. We analyze the costs of this approach and propose simple modifications to HTTP that, while interoperating with unmodified implementations, avoid the unnecessary network costs. We implemented our modifications, and our measurements show that they dramatically reduce latencies.

1. Introduction

People use the World Wide Web because it gives quick and easy access to a tremendous variety of information in remote locations. Users do not like to wait for their results; they tend to avoid or complain about Web pages that take a long time to retrieve. That is, users care about Web latency.

Perceived latency comes from several sources. Web servers can take a long time to process a request, especially if they are overloaded or have slow disks. Web clients can add delay if they do not quickly parse the retrieved data and display it for the user. Latency caused by client or server slowness, however, can in principle be solved simply by buying a faster computer, or faster disks, or more memory.

Web retrieval delay also depends on network latency. The Web is useful precisely because it provides remote access, and transmission of data across a distance takes time. Some of this delay depends on bandwidth; one cannot retrieve a 1 Mbyte file across a 1 Mbit/sec link in less than 8 seconds. You can in principle reduce this time by buying a higher-bandwidth link. But much of the latency seen by Web users comes from propagation delay: the speed of light is a constant. You cannot send even a single bit of information over, say, 3000 miles in less than 16 msec, no matter how much money you have.

In practice, most retrievals over the World Wide Web result in the transmission of relatively small amounts of data. (An unscientifically chosen sample of 200,000 HTTP retrievals shows a mean size of 12925 bytes and a median size of just 1770 bytes; excluding 12727 zero-length retrievals, the mean was 13767 bytes and the median 1946 bytes.) This means that bandwidth-related delay may not account for much of the perceived latency. For example, transmission of 20 Kbytes over a T1 (1.544 Mbit/sec) link should take about 100 msec. For comparison, the best-case small-packet round-trip time (RTT) over a coast-to-coast (US) Internet path is about 70 msec; at least half of this delay depends on the speed of light and is therefore intrinsic. When the network path is congested, queueing delays can increase the RTT by large factors.

This means that, in order to avoid network latency, we must avoid round trips through the network. Unfortunately, the Hypertext Transport Protocol (HTTP) [1], as it is currently used in the Web, incurs many more round trips than necessary.

In this paper, we analyze that problem, and show that almost all of the unnecessary round trips may be eliminated by surprisingly simple changes to the HTTP protocol and its implementations. We then present results measured using prototype implementations, which confirm that our changes result in significantly improved response times.

During the course of our work, Simon Spero published an analysis of HTTP [10], which reached conclusions similar to ours. However, we know of no other project, besides our own, that has implemented the consequent modifications to HTTP, or that has quantified the results.

2. HTTP protocol elements

We briefly sketch the HTTP protocol, to provide sufficient background for understanding the rest of this paper. We omit a lot of detail not directly relevant to HTTP latency.

The HTTP protocol is layered over a reliable bidirectional byte stream, normally TCP [8]. Each HTTP interaction consists of a request sent from the client to the server, followed by a response sent from the server to the client. Requests and responses are expressed in a simple ASCII format.

The precise specification of HTTP is in a state of flux. Most existing implementations conform to [1], a document which effectively no longer exists. A revision of the specification is in progress.

An HTTP request includes several elements: a *method* such as GET, PUT, POST, etc.; a Uniform Resource Locator (URL); a set of Hypertext Request (HTRQ) headers, with which the clients specifies things such as the kinds of documents it is willing to accept, authentication information, etc; and an optional Data field, used with certain methods such as PUT.

The server parses the request, then takes action according to the specified method. It then sends a response to the client, including a status code to indicate if the request succeeded, or if not, why not; a set of object headers, meta-information about the ‘‘object’’ returned by the server, optionally including the ‘‘content-length’’ of the response; and a Data field, containing the file requested, or the output generated by a server-side script.

Note that both requests and responses end with a Data field of arbitrary length. The HTTP protocol specifies three possible ways to indicate the end of the Data field, in order of declining priority:

1. If the ‘‘Content-Length’’ field is present, it indicates the size of the Data field and hence the end of the message.
2. The ‘‘Content-Type’’ field may specify a ‘‘boundary’’ delimiter, following the syntax for MIME multipart messages [2].
3. The server (but not the client) may indicate the end of the message simply by closing the TCP connection after the last data byte.

Later on we will explore the implications of the message termination mechanism.

3. Message and packet exchanges in HTTP

We now look at the way that the interaction between HTTP clients and servers appears on the network, with particular emphasis on the how this affects latency.

Figure 3-1 depicts the exchanges at the beginning of a typical interaction, the retrieval of an HTML document with at least one uncached inlined image. In this figure, time runs down the page, and long diagonal arrows show packets sent from client to server or vice versa. These arrows are marked with TCP packet types; note that most of the packets carry acknowledgements, but the packets marked ACK carry *only* an acknowledgement and no new data. FIN and SYN packets in this example never carry data, although in principle they sometimes could.

Shorter, vertical arrows show local delays at either client or server; the causes of these delays are given in italics. Other client actions are shown in roman type, to the left of the Client timeline.

Also to the left of the Client timeline, horizontal dotted lines show the ‘‘mandatory’’ round trip times (RTTs) through the network, imposed by the combination of the HTTP and TCP protocols. These mandatory round-trips result from the dependencies between various packet exchanges, marked with solid arrows. The packets shown with gray arrows are required by the TCP protocol, but do not directly affect latency because the receiver is not required to wait for them before proceeding with other activity.

The mandatory round trips are:

1. The client opens the TCP connection, resulting in an exchange of SYN packets as part of TCP’s three-way handshake procedure.
2. The client transmits an HTTP request to the server; the server may have to read from its disk to fulfill the request, and then transmits the response to the client. In this example, we assume that the response is small enough to fit into a single data packet, although in practice it might not. The server then closes the TCP connection, although if it has sent a Content-length field, the client need not wait for the connection termination before continuing.

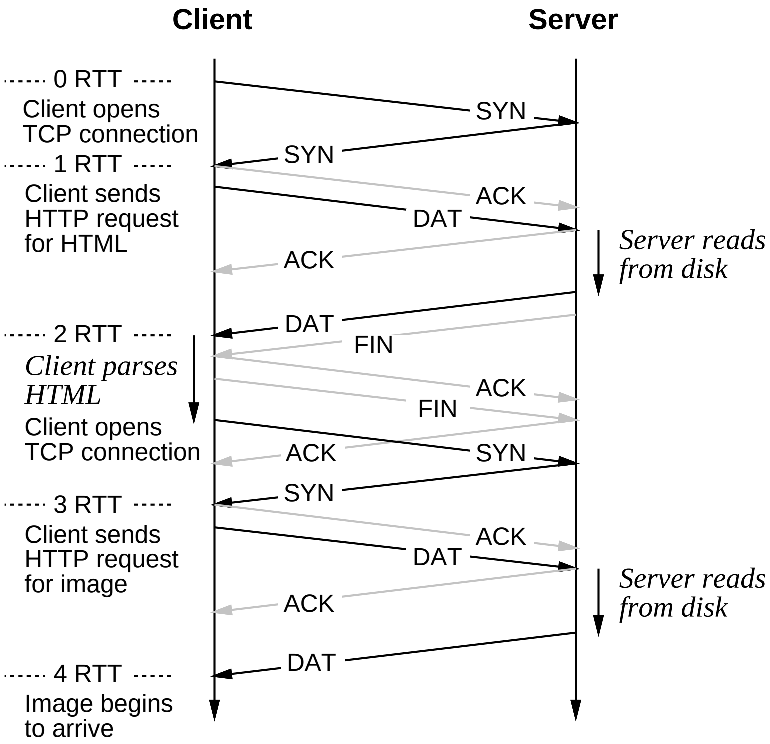

Figure 3-1: Packet exchanges and round-trip times for HTTP

3. After parsing the returned HTML document to extract the URLs for inlined images, the client opens a new TCP connection to the server, resulting in another exchange of SYN packets.

4. The client again transmits an HTTP request, this time for the first inlined image. The server obtains the image file, and starts transmitting it to the client.

Therefore, the earliest time at which the client could start displaying the first inlined image would be four network round-trip times after the user requested the document. Each additional inlined image requires at least two further round trips. In practice, with networks of finite bandwidth or documents larger than can fit into a small number of packets, additional delays will be encountered.

3.1. Other inefficiencies

In addition to requiring at least two network round trips per document or inlined image, the HTTP protocol as currently used has other inefficiencies.

Because the client sets up a new TCP connection for each HTTP request, there are costs in addition to network latencies:

- Connection setup requires a certain amount of processing overhead at both the server and the client. This typically includes allocating new port numbers and resources, and creating the appropriate data structures. Connection teardown also requires some processing time, although perhaps not as much.

- The Web clearly needs some form of authentication, and perhaps also encryption for privacy and data integrity. It would be quite expensive to re-authenticate principals on each HTTP request.

- Although the TCP connections may be active for only a few seconds, the TCP specification requires that the host which closed the connection remember certain per-connection information for four minutes [8] (although many implementations do violate this specification and use a much shorter timer.) A busy server could end up with its tables full of connections in this ‘‘TIME-WAIT’’ state, either leaving no room for new connections, or perhaps imposing excessive connection table management costs.

Current HTTP practice also means that most of these TCP connections carry only a few thousand bytes of data. As we noted earlier, one sample showed a mean document size of about 13K bytes, and a median of under 2K bytes. About 45% of these retrievals were for Graphics Interchange Format [4] (GIF) files, used for both inline and out-of-line images. This sub-sample showed a slightly larger mean and a slightly smaller median; our guess is that the very large GIF files were not inlined images. The proposed use of JPEG for inlined images will tend to reduce these sizes.

Unfortunately, TCP does not fully utilize the available network bandwidth for the first few round-trips of a connection. This is because modern TCP implementations use a technique called *slow-start* [6] to avoid network congestion. The slow-start approach requires the TCP sender to open its ‘‘congestion window’’ gradually, doubling the number of packets each round-trip time. TCP does not reach full throughput until the effective window size is at least the product of the round-trip delay and the available network bandwidth. This means that slow-start restricts TCP throughput, which is good for congestion avoidance but bad for short-connection completion latency.

3.2. Quantifying TCP connection overheads

We performed a set of simple experiments to illustrate this effect. We used a simple client program, which opens a connection to a server, tells the server how many bytes it wants, and then reads and discards that many bytes from the server. The server, meanwhile, generates the requested number of bytes from thin air, writes them into the connection, and then closes the connection. This closely approximates the network activity of a single-connection HTTP exchange.

We measured three configurations: a ‘‘local’’ server, with a round-trip time of under 1 msec, and 1460-byte TCP segments (packets); a ‘‘remote’’ server, on the other side of the country, with a best-case RTT of 70 msec; and the same remote server, but using 536-byte TCP segments. This last configuration reflects a widely-used technique meant to avoid IP fragmentation [3]; more modern practice could use the full available packet size [7].

In each configuration, we measured throughput for a large variety of connection lengths and a few popular TCP buffer (maximum window) sizes. We did ten trials for each set of parameters, and plotted the throughput of the best trial from each set (to help eliminate noise from other users of the network). Figure 3-2 shows the results for the remote (70 msec) server; figure 3-3 shows the local-server results. Note that the two figures have different vertical scales.

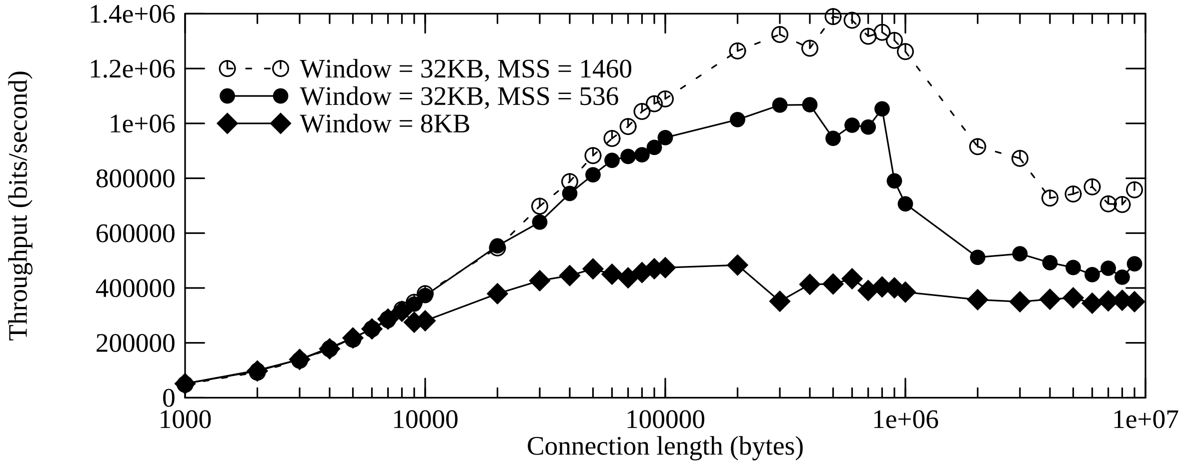

Figure 3-2: Throughput vs. connection length, RTT = 70 msec

Figure 3-2 shows that, in the remote case, using a TCP connection to transfer only 2 Kbytes results in a throughput less than 10% of best-case value. Even a 20 Kbyte transfer achieves only about 50% of the throughput available with a reasonable window size. This reduced throughput translates into increased latency for document retrieval. The figure also shows that, for this 70 msec RTT, use of too small a window size limits the throughput no matter how many bytes are transferred.

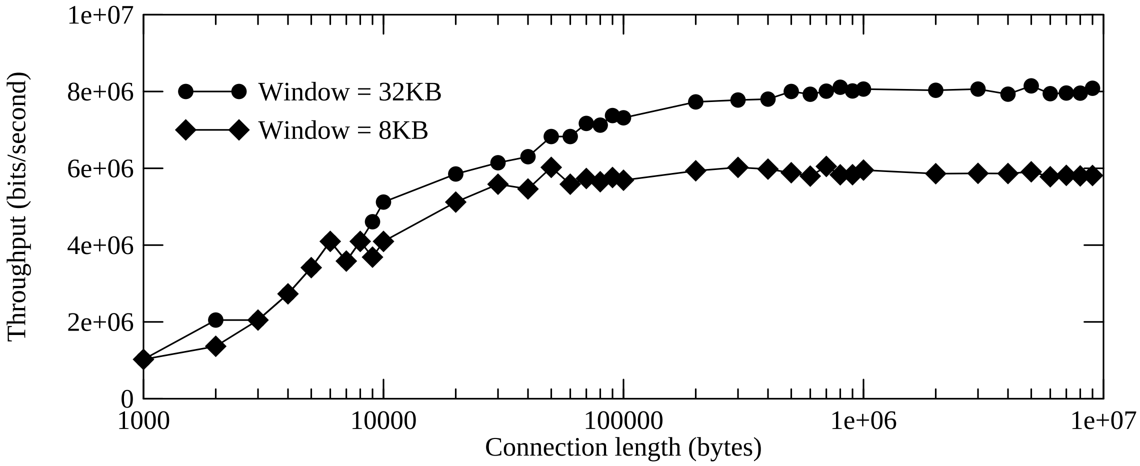

Figure 3-3: Throughput vs. connection length, RTT near 0 msec

We also note a significant decline in throughput over this path for transfers longer than about 500 Kbytes. This is caused by a breakdown in the TCP congestion-avoidance algorithm, as the congestion window becomes larger than the router's queue limit. Note, however, that this problem arises only for transfers orders of magnitude larger than typical HTML documents or inlined images. The dotted curve shows that by using a larger MSS (and hence few packets for the same congestion window size), we can obtain somewhat better throughput for lengthy transfers.

Even in the local case, per-connection overhead limits throughput to about 25% of capacity for transfers of 2 Kbytes, and about 70% of capacity for transfers of 20 Kbytes. In this case, slow-start is not involved, because the ULTRIX™ implementation of TCP avoids slow-start for local-net connections.

4. Long-lived Connections

Since the short lifetimes of HTTP connections causes performance problems, we tried the obvious solution: use a single, long-lived connection for multiple HTTP transactions. The connection stays open for all the inlined images of a single document, and across multiple HTML retrievals. This avoids almost all of the per-connection overhead, and also should help avoid the TCP slow-start delays.

Figure 4-1 shows how this change affects the network latencies. This depicts the same kind of retrieval as did figure 3-1, except that the client already has a TCP connection open to the server, and does not close it at the end of an HTTP exchange. Note that the first image arrives after just two round trips, rather than four. Also, the total number of packets is much smaller, which should lead to lower server load. Finally, since the ratio of connection lifetime to the length of the TIME-WAIT state is higher, the server will have far fewer TCP connections (active or inactive) to keep track of.

In order to use long-lived connections, we had to make simple changes to the behavior of both client and server. The client can keep a set of open TCP connections, one for each server with which it has recently communicated; it can close connections as necessary to limit its resource consumption. Even a client capable of maintaining only one open connection can benefit, by simply not closing the connection until it needs to contact a different server. It is quite likely that two successive HTTP interactions from a single client will be directed to the same server (although we have not yet quantified this locality).

The server also keeps a set of open TCP connections. Some HTTP servers fork a new process to handle each new HTTP connection; these simply need to keep listening for further requests on the open connection after responding to a request, rather than closing the connection and terminating. This not only avoids connection overhead on each request; it also avoids the cost of forking a new process for each request. Other servers manage multiple threads within a single process; these need to keep a set of TCP connections open, and listen for new requests on all of them at once. Neither approach is especially hard to implement.

With either approach, the server may need to limit the number of open connections. For example, it could close the oldest connection when the number of open connections exceeds a threshold (preferably not in the middle of

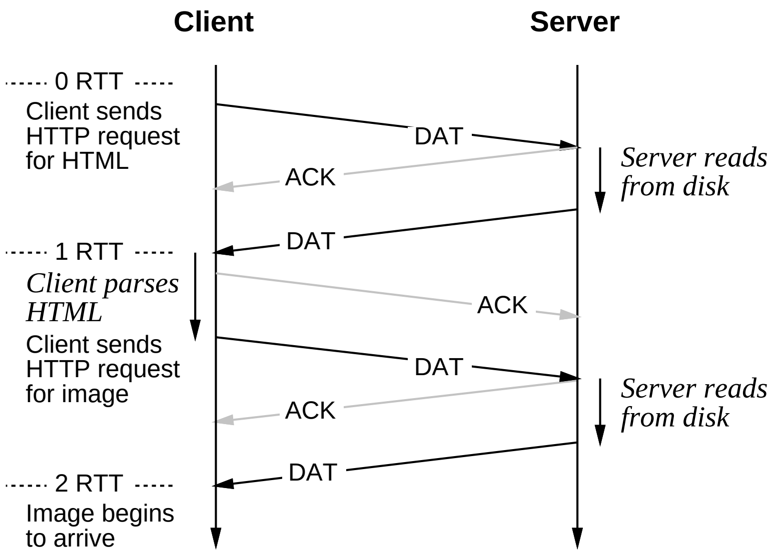

Figure 4-1: Packet exchanges for HTTP with long-lived connections

responding to a request on this connection). For multiple-process UNIX®-based servers, for example, the parent process could send its oldest child a signal (interrupt) saying ‘‘exit when you next become idle.’’ Since servers may terminate connections at arbitrary times, clients must be able to reopen connections and retry requests that fail because of this.

4.1. Detecting end-of-transmission

As we mentioned in section 2, HTTP provides three ways for the server to indicate the end of the Data field of its responses: a Content-length field, a boundary delimiter specified in the Content-type field, or termination of the TCP connection. This presents a problem when the response is generated by a script, since then the server process does not know how long the result will be (and so cannot use Content-length), nor does it know the format of the data (and so cannot safely use a predetermined delimiter sequence).

We considered several approaches in which the data stream from the script is passed through the server on its way to the client:

Boundary delimiter
The server can safely insert a boundary delimiter (perhaps as simple as a single character) if it can examine the entire data stream and ‘‘escape’’ any instance of the delimiter that appears in the data (as is done in the Telnet protocol [9]). This requires both the server and client to examine each byte of data, which is clearly inefficient.

Blocked data transmission protocol
The server could read data from the script and send it to the client in arbitrary-length blocks, each preceded by a length indicator. This would not require byte-by-byte processing, but it would involve a lot of extra data copying on the server, and would also require a protocol change.

Store-and-forward
The server can read the entire output of the script into temporary storage, then measure the length and generate a response with a correct Content-length field. This requires extra copying and may be infeasible for large responses, but does not require a protocol change.

None of these approaches appealed to us, because they all imposed extra work on the server (and possibly the client).

We also considered using a separate control connection, as in FTP, via which the server could notify the client of the amount of data it had transmitted on the data connection. This, however, might be hard to implement and doubles the amount of connection overhead, even in cases where it is not needed.

We chose to stick with a simple, hybrid approach in which the server keeps the TCP connection open in those cases where it can use the Content-length or boundary delimiter approaches, and closes the connection in other cases (typically, when invoking scripts). In the common case, this avoids the costs of extra TCP connections; in the less usual case, it may require extra connection overhead but does not add data-touching operations on either server or client, and requires no protocol changes.

4.2. Compatibility with older versions of HTTP

We wanted our modified client to transparently interoperate with both standard and modified HTTP servers, and we wanted our modified server to interoperate with both sorts of clients. This means that the modified client has to inform the server that the TCP connection should be retained, and in such a way that an unmodified server can ignore the request. This could be done by introducing a new field in the HTRQ headers (see section 2) sent in the client's request. For example, a future version of the HTTP specification could define a `hold-connection` pragma.

For our experiments, we simply encoded this information in a new HTRQ header field; such unrecognized fields must be ignored by unmodified servers.

5. Pipelining requests

Even with long-lived TCP connections, simple implementations of the HTTP protocol still require at least one network round trip to retrieve each inlined image. The client interacts with the server in a stop-and-wait fashion, sending a request for an inlined image only after having received the data for the previous one.

There is no need for this, since the retrieval of one image in no way depends on the retrieval of previous images. We considered several ways in which client requests could be pipelined, to solve this problem.

5.1. The GETALL method

When a client does a GET on a URL corresponding to an HTML document, the server just sends back the contents of the corresponding file. The client then sends separate requests for each inlined image. Typically, however, most or all of the inlined images reside on the same site as the HTML document, and will ultimately come from the same server.

We propose adding to HTTP a GETALL method, specifying that the server should return an HTML document and all of its inlined images residing on that server. On receiving this request, the server parses the HTML file to find the URLs of the images, then sends back the file and the images in a single response. The client uses the Content-length fields to split the response into its components.

The parsing of HTML documents is additional load for the server. However, it should not be too expensive, especially compared to the cost of parsing many additional HTTP requests. Or, the server could keep a cache of the URLs associated with specific HTML files, or even a precomputed database.

One can implement the GETALL method using an ordinary GET, using an additional field (such as a Pragma) in the HTRQ header to indicate that the client wants to perform a GETALL. This allows a modified client to interoperate with an unmodified server; in this case, the client notes that it has not received all the images when the connection is closed, and simply retrieves them the traditional way.

5.2. The GETLIST method

HTTP clients typically cache recently retrieved images, to avoid unnecessary network interactions. A server has no way of knowing which of the inlined images in a document are in the client's cache. Since the GETALL method causes the server to return all the images, this seems to defeat the purpose of the client's image cache (or of a caching relay [5]). GETALL is still useful in situations where the client knows that it has no relevant images cached (for example, if its cache contains no images from the server in question).

Therefore, we propose adding a GETLIST mechanism, allowing a client to request a set of documents or images from a server. A client can use a GET to retrieve an HTML file, then use the GETLIST mechanism to retrieve in one exchange all the images not in its cache. (On subsequent accesses to the same HTML file, the client can request the HTML and all images in one message.)

Logically, a GETLIST is the same as a series of GETs sent without waiting for the previous one to complete. We in fact chose to implement it this way, since it requires no protocol change and it performs about the same as an explicit GETLIST would.

Our client uses implements a simple heuristic to decide between using GETALL and GETLIST. When it accesses a document for the first time, it uses GETALL, even though there is a small chance that its cache contains some of the inlined images. It keeps a cache listing for each known image URLs the URL of the document that contained it, so the client can distinguish between documents for which it definitely has cached images, and those for which it probably does not (some images may be referenced by several documents). We have not done sufficient studies of actual HTTP usage to determine if this heuristic results in excessive retrievals of cached images.

6. Experimental Results

In this section, we report on simple experiments to measure the effect of the new protocol on observed latency.

We implemented our protocol changes by modifying the Mosaic V2.4 client, and the NCSA *httpd* V1.3 server. Both client and server were run on MIPS-based DECstation™ systems, running the ULTRIX™ operating system. The Mosaic client ran on DECstation 3100 with 24M bytes of memory; this is a relatively slow system and so we measured network retrieval times, not including the time it took to render images on the display.

In our experiments, we measured the time required to load a document and all of its inlined images. We created documents with different numbers of inlined images, and with images of various sizes. We did these measurements for both a local server, accessed via a 10 Mbit/sec Ethernet with a small RTT, and a remote server, access via a 1.544 Mbit/sec T1 link with a best-case RTT of about 70 msec.

Figure 6-1 shows how load time depends on the number of images retrieved, using 2544-byte images and the remote server. Our modified HTTP protocol cuts the latency by more than half, about what we expected from the reduced number of round trips. These images are about the median size observed in our traces, and so we do expect to see this kind of speedup in practice. While more than half of the improvement comes from pipelining, even without pipelining long-lived connections do help.

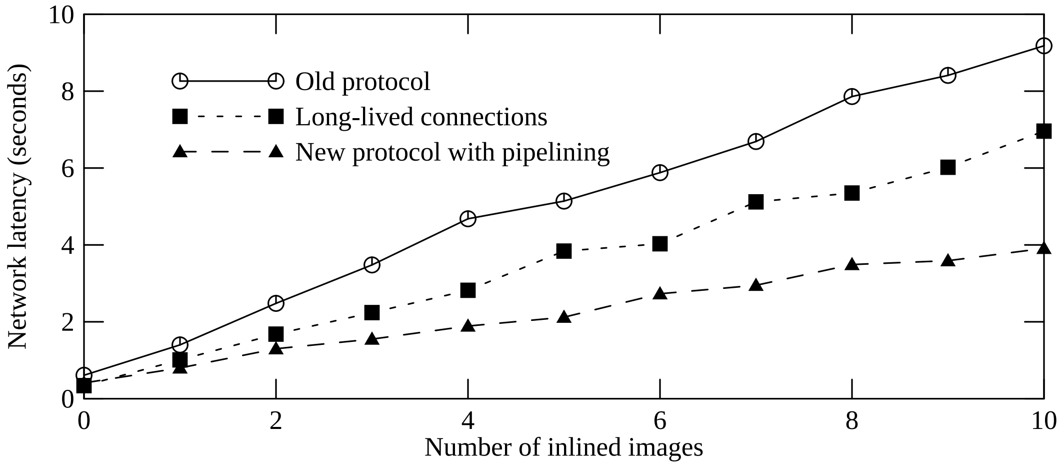

Figure 6-1: Latencies for a remote server, image size = 2544 bytes

Figure 6-2 shows that load time depends on the number of images retrieved. In this case, using 45566-byte images and the remote server, the new protocol improves latency by about 22%; less than in figure 6-1, but still noticeable. In this case, the actual data transfer time begins to dominate the connection setup and slow-start latencies.

We summarize our results for trials using the remote server and various image sizes in figure 6-3, and using the local server in figure 6-4. These graphs show the relative improvement from the modified protocol, including

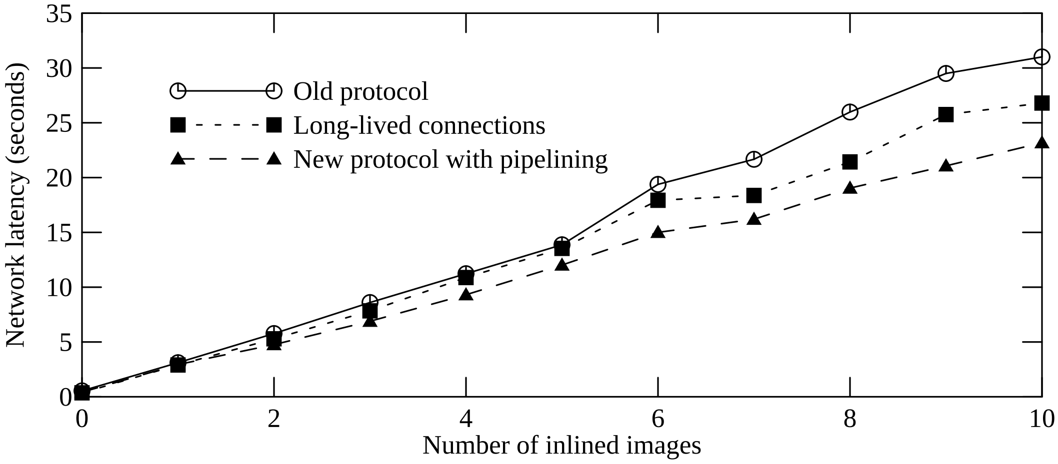

Figure 6-2: Latencies for a remote server, image size = 45566 bytes

pipelining. In general, the benefit from the modified protocol is greatest for small images and for at least a moderate number of images.

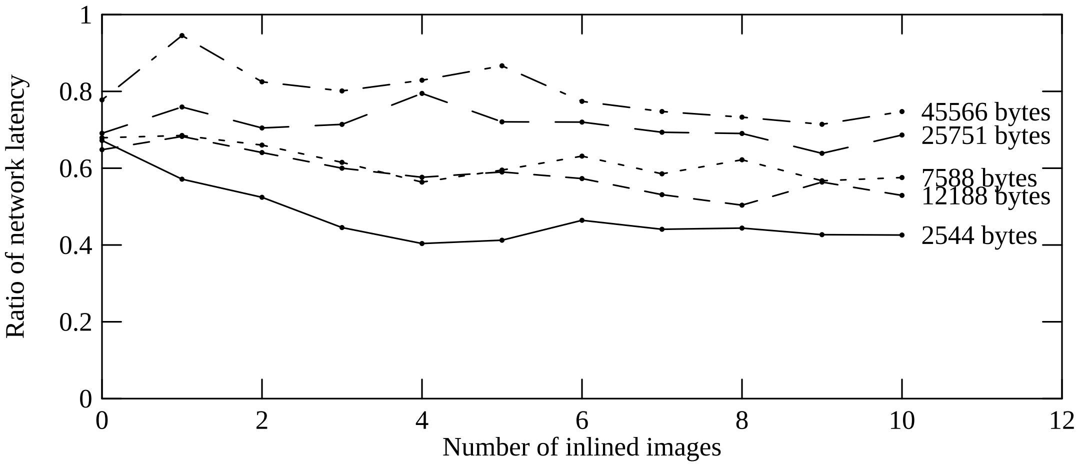

Figure 6-3: Latency improvements for a remote server

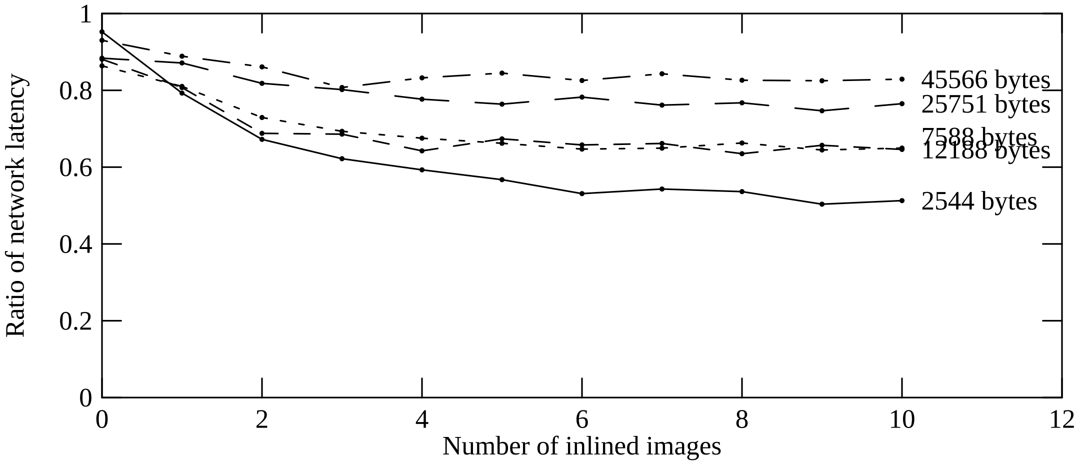

Figure 6-4: Latency improvements for a local server

Even though the round-trip time to the local server is much smaller than that to the remote server, the modified protocol still provides significant improvements for local transactions. For the local case, long-lived connections without pipelining reduces latency by only about 5% to 15%; this implies that the reduction in round trips is more important that the per-connection overheads.

Note that for the relatively small transfers associated with the median image size, slow-start latencies cannot account for much of the delay; in these tests, the TCP MSS was 1460 bytes, and traces showed that slow-start did not limit the window size.

7. Future work

The use of long-lived connections will change the mean lifetime and number of active connections (and the number of TIME-WAIT connections) at a busy HTTP server. It should also reduce the number of process creations done by multiple-process servers, at the cost perhaps of increased memory use for idle processes. We need to do further studies, using actual access patterns, to measure how these changes will affect server performance under load.

8. Conclusions

We have analyzed and quantified several sources of significant latency in the World Wide Web, problems which are inherent in the way HTTP is currently used. We have proposed several simple changes in HTTP that, individually or together, substantially reduce latency, while interoperating with unmodified servers and clients. These changes may also help reduce server loading.

Acknowledgements

We would like to thank Digital's Western Research Lab, Cambridge Research Lab, Systems Research Center, and Network Systems Lab for their help and resources. We particularly thank Jeff Kellem, Glenn Trewitt, Steve Glassman, and Paul Flaherty.

References

[1] Tim Berners-Lee. *Hypertext Transfer Protocol (HTTP)*. Internet Draft draft-ietf-iiir-http-00.txt, IETF, November, 1993. This is a working draft.

[2] N. Borenstein and N. Freed. *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. RFC 1521, Internet Engineering Task Force, September, 1993.

[3] R. Braden. *Requirements for Internet Hosts -- Communication Layers*. RFC 1122, Internet Engineering Task Force, October, 1989.

[4] CompuServe, Incorporated. Graphics Interchange Format Standard. 1987.

[5] Steven Glassman. A Caching Relay for the World Wide Web. In *Proceedings of the First International World-Wide Web Conference*, pages 69-76. Geneva, May, 1994.

[6] Van Jacobson. Congestion Avoidance and Control. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 314-329. Stanford, CA, August, 1988.

[7] Jeffrey C. Mogul and Stephen Deering. *Path MTU Discovery*. RFC 1191, Network Information Center, SRI International, November, 1990.

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

[9] J. Postel and J. Reynolds. *Telnet Protocol Specification*. RFC 854, Network Information Center, SRI International, May, 1983.

[10] Simon E. Spero. Analysis of HTTP Performance problems. URL http://elanor.oit.unc.edu/http-prob.html, July, 1994.

Venkata Padmanabhan is pursuing a PhD. in the Computer Science Department of the University of California at Berkeley. He obtained his Bachelor of Technology in Computer Science at the Indian Institute of Technology (Delhi) in 1993. His research interests include computer networks and operating systems.

Jeffrey C. Mogul received an S.B. from the Massachusetts Institute of Technology in 1979, an M.S. from Stanford University in 1980, and his PhD from the Stanford University Computer Science Department in 1986. Jeff has been an active participant in the Internet community, and is the author or co-author of several Internet Standards. Since 1986, he has been a researcher at the Digital Equipment Corporation Western Research Laboratory, working on network and operating systems issues for high-performance computer systems. He is a member of ACM, Sigma Xi, the IEEE Computer Society, and CPSR, is an associate editor of the journal *Internetworking: Research and Experience*, and was Program Committee Chair for the Winter 1994 USENIX Technical Conference. He may be reached as mogul@wrl.dec.com.

Contact author: Venkata Padmanabhan (padmanab@cs.berkeley.edu)

Table of Contents

List of Figures