The Case for Persistent-Connection HTTP

Jeffrey C. Mogul
Digital Equipment Corporation Western Research Laboratory
250 University Avenue, Palo Alto, CA 94301
mogul@wrl.dec.com

Abstract

The success of the World-Wide Web is largely due to the simplicity, hence ease of implementation, of the Hypertext Transfer Protocol (HTTP). HTTP, however, makes inefficient use of network and server resources, and adds unnecessary latencies, by creating a new TCP connection for each request. Modifications to HTTP have been proposed that would transport multiple requests over each TCP connection. These modifications have led to debate over their actual impact on users, on servers, and on the network. This paper reports the results of log-driven simulations of several variants of the proposed modifications, which demonstrate the value of persistent connections.

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. 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 can in principle be solved simply by buying a faster computer, or faster disks, or more memory.

The main contributor to Web latency, however, is network communication. 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; you can reduce this delay by buying a higher-bandwidth link. But much of the latency seen by Web users comes from propagation delay, and you cannot improve propagation delay (past a certain point) no matter how much money you have. While caching can help, many Web access are ‘‘compulsory misses.’’

If we cannot increase the speed of light, we should at least minimize the number of network round-trips required for an interaction. The Hypertext Transfer Protocol (HTTP) [3], as it is currently used in the Web, incurs many more round trips than necessary (see section 2).

Several researchers have proposed modifying HTTP to eliminate unnecessary network round-trips [21, 27]. Some people have questioned the impact of these proposals on network, server, and client performance. This paper reports on simulation experiments, driven by traces collected from an extremely busy Web server, that support the proposed HTTP modifications. According to these simulations, the modifications will improve user's perceived performance, network loading, and server resource utilization.

The paper begins with an overview of HTTP (section 2) and an analysis of its flaws (section 3). Section 4 describes the proposed HTTP modifications, and section 5 describes some of the potential design issues of the modified protocol. Section 7 describes the design of the simulation experiments, and section 8 describes the results.

2. Overview of the HTTP protocol

The HTTP protocol [1, 3] is layered over a reliable bidirectional byte stream, normally TCP [23]. 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. Request and response parameters are expressed in a simple ASCII format (although HTTP may convey non-ASCII data).

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 client 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, and takes action according to the specified method. It then sends a response to the client, including (1) a status code to indicate if the request succeeded, or if not, why not; (2) a set of object headers, meta-information about the ‘‘object’’ returned by the server; and (3) a Data field, containing the file requested, or the output generated by a server-side script.

URLs may refer to numerous document types, but the primary format is the Hypertext Markup Language (HTML) [2]. HTML supports the use of hyperlinks (links to other documents). HTML also supports the use of inlined images, URLs referring to digitized images (usually in the Graphics Interchange Format (GIF) [7] or JPEG format), which should be displayed along with the text of the HTML file by the user's browser. For example, if an HTML page includes a corporate logo and a photograph of the company's president, this would be encoded as two inlined images. The browser would therefore make three HTTP requests to retrieve the HTML page and the two images.

3. Analysis of HTTP's inefficiencies

I now analyze the way that the interaction between HTTP clients and servers appears on the network, with emphasis on 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.

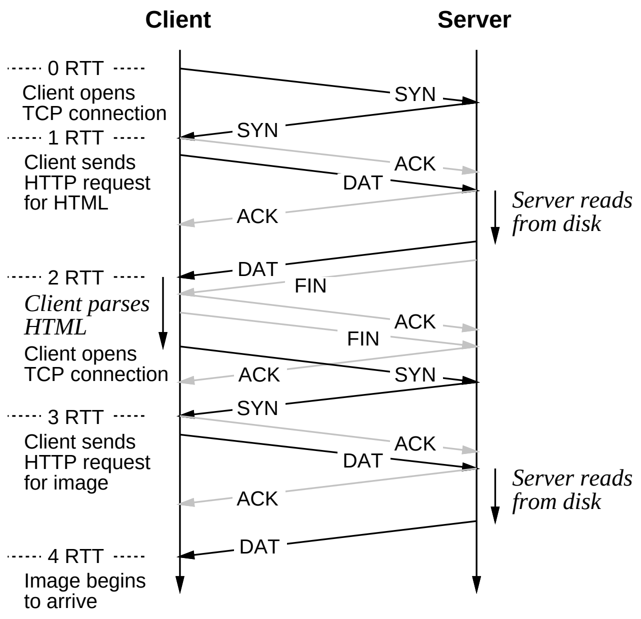

Figure 3-1: Packet exchanges and round-trip times for HTTP

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 usually the client need not wait for the connection termination before continuing.
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, for 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 TCP connections may be active for only a few seconds, but the TCP specification requires that the host that closes the connection remember certain per-connection information for four minutes [23] (Many implementations 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. I looked at retrieval size distributions for two different servers. In one, the mean size of 200,000 retrievals was 12,925 bytes, with a median of 1,770 bytes (ignoring 12,727 zero-length retrievals, the mean was 13,767 bytes and the median was 1,946 bytes). In the other, the mean of 1,491,876 retrievals was 2,394 bytes and the median 958 bytes (ignoring 83,406 zero-length retrievals, the mean was 2,535 bytes, the median 1,025 bytes). In the first sample, 45% of the retrievals were for GIF files; the second sample included more than 70% GIF files. The increasing use of JPEG images will tend to reduce image sizes.

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* [13] 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. A long-distance TCP connection may have to transfer tens of thousands of bytes before achieving full bandwidth.

4. Proposed HTTP modifications

The simplest change proposed for the HTTP protocol is to use one TCP connection for multiple requests. These requests could be for both inlined images and independent Web pages. A client would open an HTTP connection to a server, and then send requests along this connection whenever it wishes. The server would send responses in the opposite direction.

This ‘‘persistent-connection’’ HTTP (P-HTTP) avoids most of the unnecessary round trips in the current HTTP protocol. For example, once a client has retrieved an HTML file, it may generate requests for all the inlined images and send them along the already-open TCP connection, without waiting for a new connection establishment handshake, and without first waiting for the responses to any of the individual requests. We call this ‘‘pipelining.’’ Figure 4-1 shows the timeline for a simple, non-pipelined example.

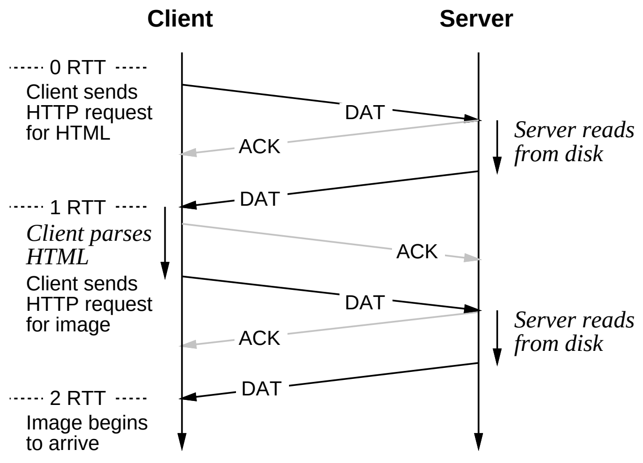

Figure 4-1: Packet exchanges and round-trip times for a P-HTTP interaction

HTTP allows the server to mark the end of a response in one of several ways, including simply closing the connection. In P-HTTP, the server would use one of the other mechanisms, either sending a ‘‘Content-length’’ header before the data, or transmitting a special delimiter after the data.

While a client is actively using a server, normally neither end would close the TCP connection. Idle TCP connections, however, consume end-host resources, and so either end may choose to close the connection at any point. One would expect a client to close a connection only when it shifts its attention to a new server, although it might maintain connections to a few servers. A client might also be ‘‘helpful’’ and close its connections after a long idle period. A client would not close a TCP connection while an HTTP request is in progress, unless the user gets bored with a slow server.

A server, however, cannot easily control the number of clients that may want to use it. Therefore, servers may have to close idle TCP connections to maintain sufficient resources for processing new requests. For example, a server may run out of TCP connection descriptors, or may run out of processes or threads for managing individual connections. When this happens, a server would close one or more idle TCP connections. One might expect a ‘‘least-recently used’’ (LRU) policy to work well. A server might also close connections that have been idle for more than a given ‘‘idle timeout,’’ in order to maintain a pool of available resources.

A server would not close a connection in the middle of processing an HTTP request. However, a request may have been transmitted by the client but not yet received when the server decides to close the connection. Or, the server may decide that the client has failed, and time out a connection with a request in progress. In any event, clients must be prepared for TCP connections to disappear at arbitrary times, and must be able to re-establish the connection and retry the HTTP request. A prematurely closed connection should not be treated as an error; an error would only be signalled if the attempt to re-establish the connection fails.

4.1. Protocol negotiation

Since millions of HTTP clients and tens of thousands of HTTP servers are already in use, it would not be feasible to insist on a globally instantaneous transition from the current HTTP protocol to P-HTTP. Neither would it be practical to run the two protocols in parallel, since this would limit the range of information available to the two communities. We would like P-HTTP servers to be usable by current-HTTP clients.

We would also like current-HTTP servers to be usable by P-HTTP clients. One could define the modified HTTP so that when a P-HTTP client contacts a server, it first attempts to use P-HTTP protocol; if that fails, it then falls back on the current HTTP protocol. This adds an extra network round-trip, and seems wasteful.

P-HTTP clients instead can use an existing HTTP design feature that requires a server to ignore HTRQ fields it does not understand. A client would send its first HTTP request using one of these fields to indicate that it speaks the P-HTTP protocol. A current-HTTP server would simply ignore this field and close the TCP connection after responding. A P-HTTP server would instead leave the connection open, and indicate in its reply headers that it speaks the modified protocol.

4.2. Implementation status

We have already published a study of an experimental implementation of the P-HTTP protocol [21]. In that paper, we showed that P-HTTP required only minor modifications to existing client and server software and that the negotiation mechanism worked effectively. The modified protocol yielded significantly lower retrieval latencies than HTTP, over both WAN and LAN networks. Since this implementation has not yet been widely adopted, however, we were unable to determine how its large-scale use would affect server and network loading.

5. Design issues

A number of concerns have been raised regarding P-HTTP. Some relate to the feasibility of the proposal; others simply reflect the need to choose parameters appropriately. Many of these issues were raised in electronic mail by members of the IETF working group on HTTP; these messages are available in an archive [12].

The first two issues discussed in this section relate to the correctness of the modified protocol; the rest address its performance.

5.1. Effects on reliability

Several reviewers have mistakenly suggested that allowing the server to close TCP connections at will could impair reliability. The proposed protocol does not allow the server to close connections arbitrarily; a connection may only be closed after the server has finished responding to one request and before it has begun to act on a subsequent request. Because the act of closing a TCP connection is serialized with the transmission of any data by server, the client is guaranteed to receive any response sent before the server closes the connection.

A race may occur between the client's transmission of a new request, and the server's termination of the TCP connection. In this case, the client will see the connection closed without receiving a response. Therefore, the client will be fully aware that the transmitted request was not received, and can simply re-open the connection and retransmit the request.

Similarly, since the server will not have acted on the request, this protocol is safe to use even with non-idempotent operations, such as the use of ''forms'' to order products.

Regardless of the protocol used, a server crash during the execution of a non-idempotent operation could potentially cause an inconsistency. The cure for this is not to complicate the network protocol, but rather to insist that the server commit such operations to stable storage before responding. The NFS specification [26] imposes the same requirement.

5.2. Interactions with current proxy servers

Many users reach the Web via ''proxy'' servers (or ''relays''). A proxy server accepts HTTP requests for any URL, parses the URL to determine the actual server for that URL, makes an HTTP request to that server, obtains the reply, and returns the reply to the original client. This technique is used to transit ''firewall'' security barriers, and may also be used to provide centralized caching for a community of users [6, 11, 22].

Section 4.1 described a technique that allows P-HTTP systems to interoperate with HTTP systems, without adding extra round-trips. What happens to this scheme if both the client and server implement P-HTTP, but a proxy between them implements HTTP [28]? The server believes that the client wants it to hold the TCP connection open, but the proxy expects the server to terminate the reply by closing the connection. Because the negotiation between client and server is done using HTRQ fields that existing proxies must ignore, the proxy cannot know what is going on. The proxy will wait ''forever'' (probably many minutes) and the user will not be happy.

P-HTTP servers could solve this problem by using an ''adaptive timeout'' scheme, in which the server observes client behavior to discover which clients are safely able to use P-HTTP. The server would keep a list of client IP addresses; each entry would also contain an ''idle timeout'' value, initially set to a small value (such as one second). If a client requests the use of P-HTTP, the server would hold the connection open, but only for the duration of the per-client idle timeout. If a client ever transmits a second request on the same TCP connection, the server would increase the associated idle timeout from the default value to a maximum value.

Thus, a P-HTTP client reaching the server through an HTTP-only proxy would encounter 1-second additional delays[1], and would never see a reply to a second request transmitted on a given TCP connection. The client could use this lack of a second reply to realize that an HTTP-only proxy is in use, and subsequently the client would not attempt to negotiate use of P-HTTP with this server.

A P-HTTP client, whether it reaches the server through a P-HTTP proxy or not, might see the TCP connection closed ''too soon,'' but if it ever makes multiple requests in a brief interval, the server's timeout would increase and the client would gain the full benefit of P-HTTP.

The simulation results in section 8 suggest that this approach should yield most of the benefit of P-HTTP. It may fail in actual use, however; for example, some HTTP-only proxies may forward multiple requests received on a single connection, without being able to return multiple replies. This would trick the server into holding the connection open, but would prevent the client from receiving all the replies.

5.3. Connection lifetimes

One obvious question is whether the servers would have too many open connections in the persistent-connection model. The glib answer is ''no, because a server could

[1]If the proxy forwards response data as soon as it is ''pushed'' by the server, then the user would not actually perceive any extra delay. This is because P-HTTP servers always indicate the end of a response using content-length or a delimiter, so the P-HTTP client will detect the end of the response even if the proxy does not.

close an idle connection at any time'' and so would not necessarily have more connections open than in the current model. This answer evades the somewhat harder question of whether a connection would live long enough to carry significantly more than one HTTP request, or whether the servers would be closing connections almost as fast as they do now.

Intuition suggests that locality of reference will make this work. That is, clients tend to send a number of requests to a server in relatively quick succession, and as long as the total number of clients simultaneously using a server is ''small,'' the connections should be useful for multiple HTTP requests. The simulations (see section 8) support this.

5.4. Server resource utilization

HTTP servers consume several kinds of resources, including CPU time, active connections (and associated threads or processes), and protocol control block (PCB) table space (for both open and TIME_WAIT connections). How would the persistent-connection model affect resource utilization?

If an average TCP connection carries more than one successful HTTP transaction, one would expect this to reduce server CPU time requirements. The time spent actually processing requests would probably not change, but the time spent opening and closing connections, and launching new threads or processes, would be reduced. For example, some HTTP servers create a new process for each connection. Measurements suggest that the cost of process creation accounts for a significant fraction of the total CPU time, and so persistent connections should avoid much of this cost.

Because we expect a P-HTTP server to close idle connections as needed, a busy server (one on which idle connections never last long enough to be closed by the idle timeout mechanism) will use up as many connections as the configuration allows. Therefore, the maximum number of open connections (and threads or processes) is a parameter to be set, rather than a statistic to be measured.

The choice of the idle timeout parameter (that is, how long an idle TCP connection should be allowed to exist) does not affect server performance under heavy load from many clients. It can affect server resource usage if the number of active clients is smaller than the maximum-connection parameter. This may be important if the server has other functions besides HTTP service, or if the memory used for connections and processes could be applied to better uses, such as file caching.

The number of PCB table entries required is the sum of two components: a value roughly proportional to the number of open connections (states including ESTABLISHED, CLOSING, etc.), and a value proportional to the number of connections closed in the past four minutes (TIME_WAIT connections). For example, on a server that handles 100 connections per second, each with a duration of one second, the PCB table will contain a few hundred entries related to open connections, and 24,000 TIME_WAIT entries. However, if this same server followed the persistent-connection model, with a mean of ten HTTP requests per active connection, the PCB table would contain only 2400 TIME_WAIT entries.

PCB tables may be organized in a number of different ways [16]. Depending on the data structures chosen, the huge number of TIME_WAIT entries may or may not affect the cost of looking up a PCB-table entry, which must be done once for each received TCP packet. Many existing systems derived from 4.2BSD use a linear-list PCB table [15], and so could perform quite badly under a heavy connection rate. In any case, PCB entries consume storage.

The simulation results in section 8 show that persistent-connection HTTP significantly reduces the number of PCB table entries required.

5.5. Server congestion control

An HTTP client has little information about how busy any given server might be. This means that an overloaded HTTP server can be bombarded with requests that it cannot immediately handle, leading to even greater overload and congestive collapse. (A similar problem afflicts naive implementations of NFS [14].) The server could cause the clients to slow down, somewhat, by accepting their TCP connections but not immediately processing the associated requests. This might require the server to maintain a very large number of TCP connections in the ESTABLISHED state (especially if clients attempt to use several TCP connections at once; see section 6).

Once a P-HTTP client has established a TCP connection, however, the server can automatically benefit from TCP's flow-control mechanisms, which prevent the client from sending requests faster than the server can process them. So while P-HTTP cannot limit the rate at which new clients attack an overloaded server, it does limit the rate at which any given client can make requests. The simulation results presented in section 8, which imply that even very busy HTTP servers see only a small number of distinct clients during any brief interval, suggest that controlling the per-client arrival rate should largely solve the server congestion problem.

5.6. Network resources

HTTP interactions consume network resources. Most obviously, HTTP consumes bandwidth, but IP also imposes per-packet costs on the network, and may include per-connection costs (e.g., for firewall decision-making). How would a shift to P-HTTP change consumption patterns?

The expected reduction in the number of TCP connections established would certainly reduce the number of ''overhead'' packets, and would presumably reduce the total number of packets transmitted. The reduction in header traffic may also reduce the bandwidth load on low-bandwidth links, but would probably be insignificant for high-bandwidth links.

The shift to longer-lived TCP connections should improve the congestion behavior of the network, by giving the TCP end-points better information about the state of the network. TCP senders will spend proportionately less time in the ''slow-start'' regime [13], and more time in the ''congestion avoidance'' regime. The latter is generally less likely to cause network congestion.

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.