HTTP Working Group
INTERNET-DRAFT
Expires September 26, 1997

J. Gettys, Digital Equipment Corporation
A. Freier, Netscape Communications Corporation

March 26, 1997

HTTP Connection Management

draft-ietf-http-connection-00.txt

1. Abstract

The HTTP/1.1 specification (RFC 2068) is silent about various details of TCP connection management when using persistent connections. This document discusses some of the implementation issues discussed during HTTP/1.1's design, and introduces a few new requirements on HTTP/1.1 implementations learned from implementation experience, not fully understood when RFC 2068 was issued. This is an initial draft for working group comment, and we expect further drafts.

2. Table of Contents

3. Key Words

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC xxxx. [Bradner]

4. Connection Management for Large Scale HTTP Systems

Recent development of popular protocols (such as HTTP, LDAP, ...) have demonstrated that the standards and engineering communities have not yet come to grip with the concept of connection management. For instance, HTTP/1.0 [HTTP/1.0] uses a TCP connection for carrying exactly one request/response pair. The simplistic beauty of that model has much less than optimal behavior.

This document focuses HTTP/1.1 implementations but the conclusions drawn here may be applicable to other protocols as well.

The HTTP/1.1 Proposed Standard [HTTP/1.1] specification is silent on when, or even if, the connection should be closed (implementation experience was desired before the specification was frozen on this topic). So HTTP has moved from a model that closed the connection after every request/response to one that might never close. Neither of these two extremes deal with "connection management" in any workable sense.

5. Resource Usage (Who is going to pay?)

The Internet is all about scale: scale of users, scale of servers, scale over time, scale of traffic. For many of these attributes, clients must be cooperative with servers.

Clients of a network service are unlikely to communicate with more than a few servers (small number of 10s). Considering the power of desktop machines of today, maintaining that many idle connections does not appear to be overly burdensome, particularly when you consider the client is the active party and doesn't really have to pay attention to the connection unless it is expecting some response.

Servers will find connections to be critical resources and will be forced to implement some algorithm to shed existing connections to make room for new ones. Since this is an area not treated by the protocol, one might expect a variety of "interesting" efforts.

Maintaining an idle connection is almost entirely a local issue. However, if that local issue is too burdensome, it can easily become a network issue. A server, being passive, must always have a read pending on any open connection. Some implementations of the multi-wait mechanisms tend to bog down as the number of connections climbs in to the hundreds, though operating system implementations can scale this into the thousands, tens of thousands, or even beyond. Whether server implementations can also scale to so many simultaneous clients is likely much less clear than if the operating system can theoretically support such use. Implementations might be forced to use fairly bizarre mechanisms, which could lead to server instability, and then perhaps service outages, which are indeed a network issues. And despite any heroic efforts, it will all be to no avail. The number of clients that could hold open a connection will undoubtedly overwhelm even the most robust of servers over time.

When this happens, the server will of necessity be forced to close connections. The most often considered algorithm is an LRU. The success of LRU algorithms in other areas of computer engineering is based on locality of reference. I.e., in this case, if LRU is better than random, then this is because the "typical" client's behavior is predictable based on its recent history. Clients that have made requests recently are probably more likely to make them again, than clients which have been idle for a while. While we are not sure we can point to rigorous proof of this principle, we believe it does hold for Web service and client reference patterns are certainly a very powerful "clue".

The client has more information that could be used to drive the process. For instance, it does not seem to much to expect that a connection be held throughout the loading of a page and all its embedded links. It could further sense user sincerity towards the page by detecting such events as mouse movement, scrolling, etc., as

indicators that there is still some interest in pursing the page's
content, and therefore the chance of accessing subsequent links. But
if the user has followed a number of links in succession away to a
different server, it may be likely that the first connection will not

be used again soon. Whether this is significantly better than LRU is an open question, but it is clear that unlikely to be used connections should be closed, to free the server resouces involved. Server resouces are much more scarce than client resources, and clients should be frugal, if the Web is to have good scaling properties.

Authoritative knowledge that it is appropriate to close a connection can only come from the user. Unfortunately, that source is not to be trusted. First, most users don't know what a connection is, and having them indicate it is okay to close it is meaningless. Second, a user that does know what a connection is probably inherently greedy. Such a user would never surrender the attention that a connection to a server implies. Research [Mogul2] does show that most of the benefits of persistent connections are gained if connections can be held open after last use approximately one minute for the HTTP traffic studied; this captures most "click ahead" behavior of a user's web browsing.

For many important services, server resources are critical resources; there are many more clients than services. For example, the AltaVista search service handles (as of this writing) tens of millions of searches per day, for millions of different clients. While it is one of the two or three most popular services on the Internet today, it is clearly small relative to future services built with Internet technology and HTTP. From this perspective, it is clear that clients need to cooperate with servers to enable servers to continue to scale.

System resources at a server:

* Server resources (open files, file system buffers, processes, memory for applications, memory for socket buffers for connections currently in use (16-64Kbytes each, data base locks). In BSD derived TCP implementations, socket buffers are only needed on active connections. This usually works because it's seldom the case that there is data queued to/from more than a small fraction of the open connections.

* PCB (Protocol control blocks, only ~100-140 bytes; even after a connection is closed, you can't free this data structure for a significant amount of time, of order minutes. More severe, however, is that many inferior TCP implementations have had linear or quadratic algorithms relating to the number of PCB's to find PCB's when needed.

These are organized from most expensive, to least.

Clients should read data from their TCP implementations aggressively, for several reasons:

* TCP implementations will delay acknowledgements if socket

buffers are not emptied. This will lower TCP performance, and cause increased elapsed time for the end user. [Frystyk et. al.] while continuing to consume the server's resources.

* Servers must be able to free the resources held on behalf of the client as quickly as possible, so that the server can reuse these resources on behalf of others. These are often the largest and scarcest server system resource (processes, open files, file system buffers, data base locks, etc.)

When HTTP requests complete (and a connection is idle), an open connection still consumes resources some of which are not under the server's control:

* socket buffers (16-64KB both in the operating system, and often similar amounts in the server process itself)

* Protocol Control Blocks (.15 KB/PCB's). (??? Any other data structures associated with PCB's?)

If, for example, an HTTP server had to indefinitely maintain these resources, this memory alone for a million clients (and there are already HTTP services larger than this scale in existence today) using a single connection each would be tens of gigabytes of memory. One of the reasons the Web has succeeded is that servers can, and do delete connections, and require clients to reestablish connections.

If connections are destroyed too aggressively (HTTP/1.0 is the classic limiting case), other problems ensue.

* The state of congestion of the network is forgotten [Jacobson]. Current TCP implementations maintain congestion information on a per-connection basis, and when the connection is closed, this information is lost. The consequences of this are well known: general Internet congestion, and poor user performance

* Round trip delays and packets to re-establish the connections. Since most objects in the Web are very small, of order half the packets in the network has been due to just the TCP open and close operation.

* Slow Start lowers initial throughput of the TCP connection

* PCB's become a performance bottleneck in some TCP implementations (and cannot be reused for a XXX timeout after the connection has been terminated). The absolute number of PCBs in the TIME_WAIT state could be much larger than the number in the ESTABLISHED state. Closing connections too quickly can actually consume more memory than closing them

slowly, because all PCBs consume memory and idle socket buffers do not.

From these two extreme examples, it is obvious that connection management becomes a central issue for both clients and servers.

Clearly, benefits of persistent connections will be lost if clients open many connections simultaneously. RFC2068 therefore specifies no more than 2 connections from a client to a server should be open at any one time, or 2N connections (where N is the number of clients a proxy is serving) for proxies. Frystyk et. al. have shown that roughly twice the performance of HTTP/1.0 using four to six connections can be reached using HTTP/1.1 over a single TCP connection using HTTP/1.1, even over a LAN, once combined with compression of the HTML documents [Frystyk].

6. Go to the Head of the Line

The HTTP/1.1 specification requires that proxies use no more than 2N connections, where N is the number of client connections being served. Mogul has shown that persistent connections are a "good thing", and Frystyk et. al. show data that significant (a factor of 2-8) savings in number of packets transmitted result by using persistent connections.

If fewer connections are better, then, why does HTTP/1.1 permit proxies to establish more than the absolute minimum of connections? In the interests of brevity, the HTTP/1.1 specification is silent on some of the motivations for some requirements of the specification. At the time HTTP/1.1 was specified, we realized that if a proxy server attempted to aggregate requests from multiple client connections onto a single TCP connection, a proxy would become vulnerable to the "head of line" blocking problem. If Client A, for example, asks for 10 megabytes of data (or asked for a dynamicly generated document of unlimited length), then if a proxy combined that request with requests from another Client B, Client B would never get its request processed. This would be a very "bad thing", and so the HTTP/1.1 specification allows proxies to scale up their connection use in proportion to incoming connections. This will also result in proxy servers getting roughly fair allocation of bandwidth from the Internet proportional to the number of clients.

Since the original HTTP/1.1 design discussions, we realized that there is a second, closely related denial of service security arises if proxies attempt to use the same TCPconnection for multiple clients. An attacker could note that a particular URL of a server that they wished to attack was either very large, very slow (script based), or never returned data. By making requests for that URL, the attacker could easily block other clients from using that server entirely, due to head of line blocking, so again, simultaneously multiplexing requests from different clients would be very bad, and therefore implementations MUST not attempt such multipexing.

In other words, head-of-line blocking couples the fates of what should be independent interactions, which allows for both denial-of-service attacks, and for accidental synchronization.

Here is another example of head-of-line blocking: imagine clients A and B are connected to proxy P1, which is connected to firewall proxy P2, which is connected to the Internet. If P1 only has one connection to P2, and A attempts to connect (via P1 and P2) to a dead server on the Internet, all of B's operations are blocked until the connection attempt from P2 to the dead server times out. This is not a good situation.

Note that serial reuse of a TCP connection does not have these considerations: a proxy might first establish a connection to an origin server for Client A, and possibly leave the connection open after Client A finishes and closes

its connection, and then use the same connection for Client B, and so on. As in normal clients, such a proxy should close idle connections.

Future HTTP evolution also dictates that simultaneous multiplexing of clients over a connection should be prohibited. A number of schemes for compactly encoding HTTP rely on associating client state with a connection, which HTTP 1.X does not currently do. If proxies do such multiplexing, then such designs will be much harder to implement.

7. The Race is On

Deleting a connection without authoritative knowledge that it will not be soon reused is a fundamental race that is part of any timeout mechanism. Depending on how the decision is made will determine the penalties imposed.

It is intuitively (and most certainly empirically) less expensive for the active (client) partner to close a connection than the server. This is due in most part to the natural flow of events. For instance, a server closing a connection cannot know that the client might at that very moment be sending a request. The new request and the close message can pass by in the night simply because the server and the client are separated by a network. That type of failure is a network issue. The code of both the client and the server must to be able to deal with such failures, but they should not have to deal with it efficiently. A client closing a connection, on the other hand, will at least be assured that any such race conditions are mostly local issues. The flow will be natural, assuming one treats closing as a natural event. To paraphrase Butler Lampson's 1983 paper on system design, "The events that happen normally must be efficient. The exceptional need to make progress." [Lampson]

Having the client closing the connection will decrease the

probability of the client having to do automatic connection recovery
of a pipeline caused by a premature close on server side. From an
client implementation point of view this is advantageous as automatic

connection recovery of a pipeline is significantly more complicated than closing an idle connection. In HTTP, however, servers are free to close connections any time, and this observation does not help, but may simplify other protocols. It will, however, reduce the number of TCP resets observed, and make the exceptional case exceptional, and avoid a TCP window full of requests being transmitted under some circumstances.

On the one hand, it is a specific fact about TCP that if the client closes the connection, the server does not have to keep the TIME_WAIT entry lying around. This is goodness.

On the other hand, if the server has the resources to keep the connection open, then the client shouldn't close it unless there is little chance that the client will use the server again soon, since closing & then reopening adds computational overhead to the server. So allowing the server to take the lead in closing connections does have some benefits.

A further observation is that congestion state of the network varies with time, so the benefits of the congestion state being maintained by TCP diminishes the longer a connection is idle.

This discussion also shows that a client should close idle connections before the server does. Currently in the HTTP standard there is no way for a server to provide such a "hint" to the client, and there should be a mechanism. This memo solicits other opinions on this topic.

8. Closing Half of the Connection

In simple request/response protocols (e.g. HTTP/1.0), a server can go ahead and close both recieve and transmit sides of its connection simultaneously whenever it needs to. A pipelined or streaming protocol (e.g. HTTP/1.1) connection, is more complex [Frystyk et. al.], and an implementation which does so can create major problems.

The scenario is as follows: an HTTP/1.1 client talking to a HTTP/1.1 server starts pipelining a batch of requests, for example 15 on an open TCP connection. The server decides 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 which makes the client TCP stack 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. Servers must therefore close each half of the connection independently.

9. Capture Effect

One of the beauties of the simple single connection for each request/response pair is that it did not favor an existing client over another. In general, this natural rotation made for a fairer offering of the overall service, albeit a bit heavy handed. Our expectation is that when protocols with persistent connections get heavily deployed, that aspect of fairness will not exist. Without some moderately complex history, it might be that only the first 1000 clients will ever be able to access a server (providing that your server can handle 1000 connections).

There needs to be some policy indicating when it is appropriate to close connections. Such a policy should favor having the client be the party to initiate the closure, but must provide some manner in which the server can protect itself from misbehaving clients. Servers can control greedy clients in HTTP/1.1 by use of the 503 (Service Unavailable) response code in concert with the Retry-After response-header field, or by not reading further requests from that client, at the cost of temporarily occupying the connection. As long as the server can afford to keep the connection open, it can delay a "greedy client" by simply closing the TCP receive window. As soon as it drops the connection, it has no way to distinguish this client from any other. Either of these techniques may in fact be preferable to closing the client's connection; the client might just immediately reopen the connection, and you are unlikely to know if it is the same greedy client.

Implementation complexity will need to be balanced against scheduling overhead. A number of possible server scheduling algorithms exist, with different costs and benefits. The implementation experience of one of us (jg) with the X Window System [Gettys et. al.] may be of use to those implementing Web server schedulers.

* Strict round robin scheduling: a operating system select or poll operation is executed for each request processed, and each request is handled in turn (across connections). Since select is executed frequently, new connections get a good chance of service sooner rather than later. Some algorithm must be chosen to avoid capture effect if the server is loaded. This is most fair, and approximates current behavior. The disadvantage is, however, a (relatively expensive) system call / request, which will likely become too expensive as Web servers become carefully optimized after HTTP/1.1 is fully implemented.

* Modified round robin scheduling: a operating system select or
  poll operation is executed. Any new connections are
  established, and for each connection showing data available,

all available requests are read into buffers for later execution. Then all requests are processed, round robin between buffers. Some algorithm must be chosen to avoid capture effect if the server is loaded. This eliminates the system call per operation. This is quite efficient, and still reasonably fairly apportions server capabilities.

* Some servers are likely to be multithreaded, possibly with a thread per connection. These servers will have to have some mechanism to share state so that no client can forever capture a connection on a busy server.

A final note: indefinite round robin scheduling may not in fact be the most desirable algorithm, due to the timesharing fallacy. If a connection makes progress more slowly than possible, not only will the client (the end user) observe poorer performance, but the connection (and the considerable system overhead each one represents) will be open longer, and more connections and server resources will be required as a result.

At some point, large, loaded servers will have to choose a connection to close; research [Padmanabhan and Mogul] shows that LRU may be as good as more complex algorithms for choosing which to close.

Further experimentation with HTTP/1.1 servers will be required to understand the most useful scheduling and connection management algorithms.

10. Security Considerations

Most HTTP related security considerations are discussed in RFC2068. This document identifies a further security concern: proxy implementations that simultaneously multiplex requests from multiple clients over a TCP connection are vulnerable to a form of denial of service attacks, due to head of line blocking problems, as discussed further above.

The capture effect discussed above also presents opportunities for denial of service attacks.

11. Requirements on HTTP/1.1 Implementations

Here are some simple observations and requirements from the above discussion.

* clients and proxies SHOULD close idle connections. Most of the benefits of an open connection diminish the longer the connection is idle: the congestion state of the network is a dynamic and changing phenomena [Paxson]. The client, better

than a server, knows when it is likely not to revisit a site. By monitoring user activity, a client can make reasonable guesses as to when a connection needs closing. Research has shown [Mogul2] shows that most of the benefits of a persistent connection are likely to occur within approximately 60 seconds. Further research in this area is needed. On the client side, define a connection as "idle" if it meets at least one of these two criteria:

  * no user-interface input events during the last 60 seconds (parameter value shouldn't be defined too precisely)

  * user has explicitly selected a URL from a different server. Don't switch just because inlined images are from somewhere else! Even in this case, dally for some seconds (e.g., 10) in case the user hits the "back" button.

On the server side, use a timeout that is adapted based on resource constraints: short timeout during overload, long timeout during underload. Memory, not CPU cycles, is likely to be the controlling resource in a well-implemented system.

* servers SHOULD implement some mechanism to avoid the capture effect.

* proxies MUST use independent TCPconnections to origin or futher proxy servers for different client connections, both to avoid head of line blocking between clients, and to avoid the denial of service attacks that implementations that attempt to multiplex multiple clients over the same connection would be open to.

* proxies MAY serially reuse connections for multiple clients.

* servers MUST properly close incoming and outgoing halves of TCP connections independently.

* clients SHOULD close connections before servers when possible. Currently, HTTP has no "standard" way to indicate idle time behavior to clients, though we note that the Apache HTTP/1.1 implementation advertizes this information using the Keep-Alive header if Keep-Alive is requested. We note, however, that Keep-Alive is NOT currently part of the HTTP standard, and that the working group may need to consider providing this "hint" to clients in the future of the standard by this or other means not currently specified in this initial draft.

12. References

[Apache]
The Apache Authors, The Apache Web Server is distributed by The Apache Group.

[Bradner]
S. Bradner, "Keywords for use in RFCs to Indicate Requirement Levels", RFC XXXX

[Frystyk]
Henrik Frystyk Nielsen, "The Effect of HTML Compression on a LAN ", W3C. URL:
http://www.w3.org/pub/WWW/Protocols/HTTP/Performance/Compression/LAN.html

[Frystyk et. al]
Henrik Frystyk Nielsen, Jim Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, W3C, H&aring;kon Wium Lie, Chris Lilley, W3C, "Network Performance Effects of HTTP/1.1, CSS1, and PNG". W3C Note, February, 1997. See URL:
http://www.w3.org/pub/WWW/Protocols/HTTP/Performance/ for this and other HTTP/1.1 performance information.

[Gettys et. al.]
Gettys, J., P.L. Karlton, and S. McGregor, " The X Window System, Version 11.'' Software Practice and Experience Volume 20, Issue No. S2, 1990 ISSN 0038-0644.

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

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

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

[Lampson]
B. Lampson, "Hints for Computer System Design", 9th ACM SOSP, Oct. 1983, pp. 33-48.

[Mogul]
Jeffrey C. Mogul. "The Case for Persistent-Connection HTTP." In

Proc. SIGCOMM '95 Symposium on Communications Architectures and Protocols, pages 299-313. Cambridge, MA, August, 1995.

[Mogul2]
Jeffrey C. Mogul. "The Case for Persistent-Connection HTTP". Research Report 95/4, Digital Equipment Corporation Western Research Laboratory, May, 1995. URL: http://www.research.digital.com/wrl/techreports/abstracts/95.4.html

[Padmanabhan and Mogul]
Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP Latency. In Proc. 2nd International WWW Conf. '94: Mosaic and the Web, pages 995-1005. Chicago, IL, October, 1994. URL: http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html

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

[Paxson]

Vern Paxson, "End-to-end Routing Behavior in the Internet" ACM SIGCOMM '96, August 1996, Stanford, CA.

13. Acknowlegements

Our thanks to Henrik Frystyk Nielsen for comments on the first draft of this document.

14. Authors' Addresses

Jim Gettys
W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA
Fax: +1 (617) 258 8682
Email: jg@w3.org

Alan Freier
Netscape Communications Corporation
Netscape Communications
501 East Middlefield Rd.
Mountain View, CA 94043
Email: freier@netscape.com