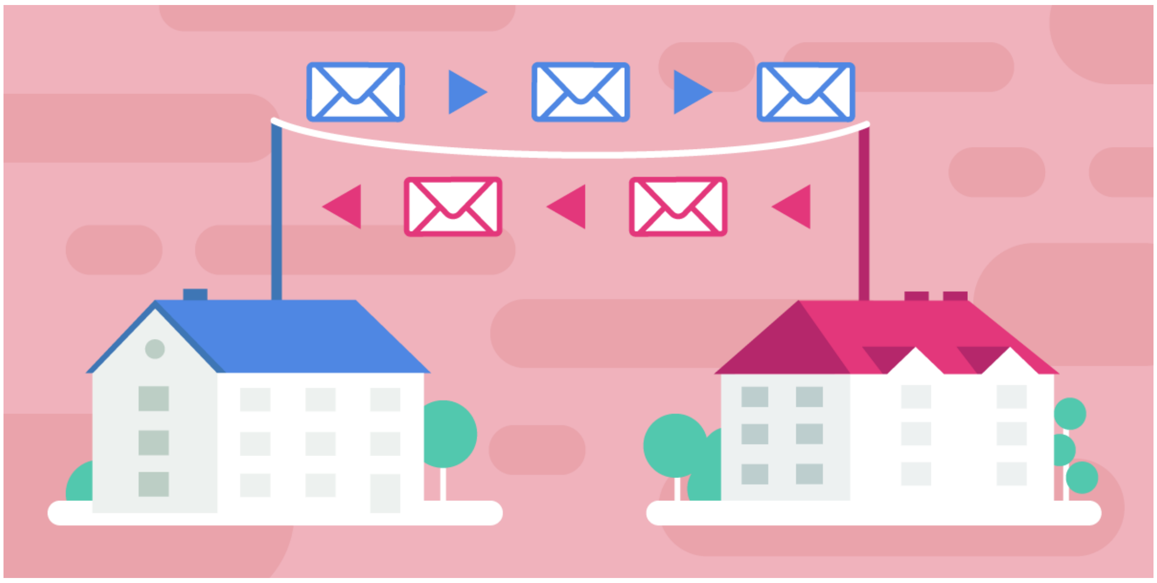
HTTP Keep-Alive, Pipelining, Multiplexing and Connection Pooling

Baptiste Assmann | Dec 15, 2020 | LOAD BALANCING / ROUTING, PERFORMANCE | 1 comment

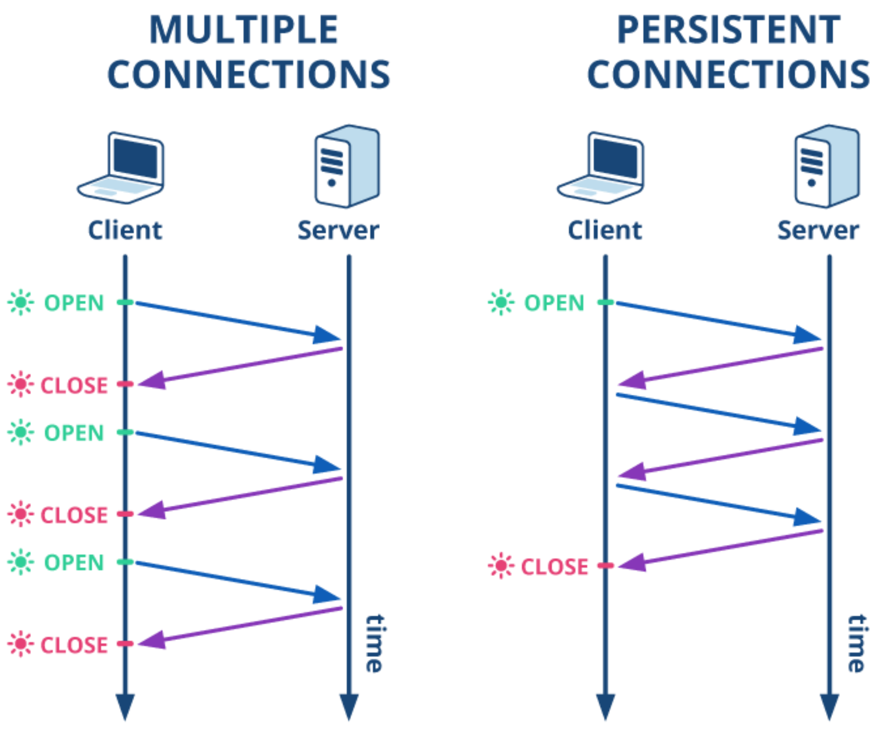


Persistent connections allow HAProxy to optimize resource usage, lower latency on both the client and server side, and support connection pooling.

HTTP is a layer 7 protocol that's transmitted over a TCP connection. It works in a client-server model and follows the request-response paradigm, which means that a client sends a request to a server, which then replies with a response.

From this statement, you can infer two different ways of operating. Either the TCP connection used for the communication between the client and the server is opened for a single request-response exchange, or it is kept open for a while and both parties can use it for multiple requests and responses.

This diagram highlights this in a simpler way:



Multiple connections vs persistent connections

The diagram above clearly shows some advantages to keeping a persistent TCP connection open for multiple HTTP requests:

- Less resource usage, especially if you use SSL between the TCP and HTTP layers, since the SSL handshake happens only once.
- Fewer network round trips because there are fewer TCP connection handshakes.
- Lower application latency because less time is spent re-establishing connections.
- More efficient usage of the TCP stack (window size, network congestion algorithm).

This blog article will focus on persistent TCP connections in an HTTP world and how HAProxy supports it.

History of Keep-alive in HTTP

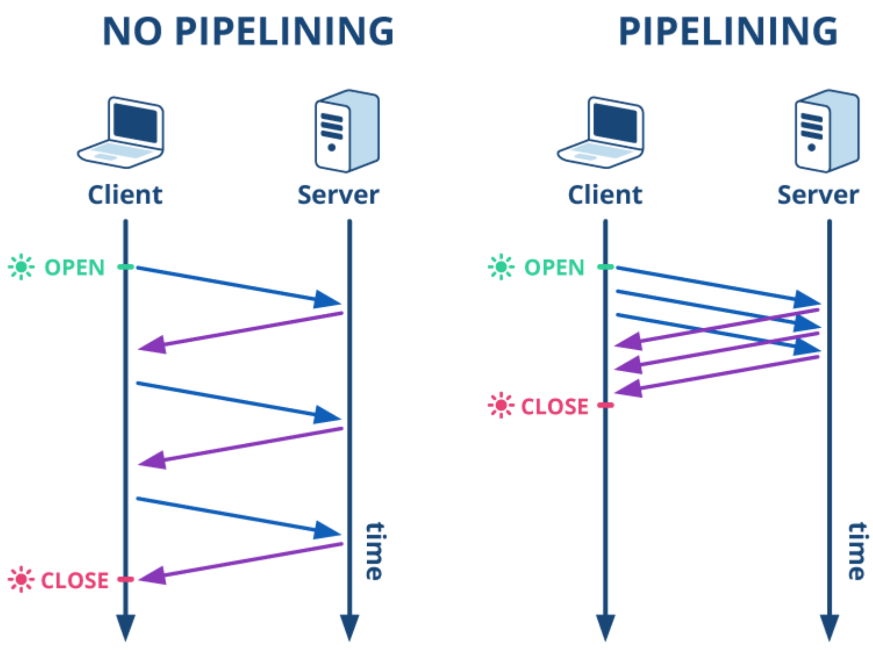
Before describing how HAProxy supports persistent connections, let's recall the history of the HTTP Keep-Alive feature and how it has evolved over time. As you may know, HTTP is a session-less protocol. Each request and response sequence is independent from each other, which means that, on its own, HTTP requires each request to have its own connection. To make it more efficient, we need HTTP Keep-Alive. HTTP Keep-alive is the mechanism that instructs the client and server to maintain a persistent TCP connection, decoupling the one-to-one relationship between TCP and HTTP, effectively increasing the scalability of the server.

The HTTP 1.0 protocol does not support persistent connections by default. In order to do so, the client has to send a *Connection* header with its value set to *keep-alive* to indicate to the server that the connection should remain open for any subsequent requests. That said, this is not a hard and fast rule and the server can close said connection after the first response—or any response actually. In short, the keep-alive mode in HTTP 1.0 is explicit. Both the client and server have to announce it.

HTTP 1.0 is very old (almost 25 years old) and is almost no longer used on the Internet.

HTTP 1.1 supports persistent connections by default. There is no need to send the *Connection* header field to announce support for keep-alive. Each party expects that the peer supports it, although it is possible for any peer of the transaction to change this behavior by sending a *Connection: close* header. A persistent connection can be closed after any full response is sent or after some time when the connection has become idle, the purpose of which is to save resources on the server side.

Because HTTP 1.1 relies on persistent connections, you can use it to send multiple queries in a row and expect responses in the same order. This is called *HTTP pipelining*, which is demonstrated in the next diagram:



HTTP pipelining

With pipelining, the browser can send multiple requests across a single persistent connection, which allows the benefits of persistent connections to flourish. However, the drawback is that requests must queue up on the connection until the requests in front of them complete. This is called *head-of-line blocking*. To download content faster from web servers, browsers usually open up to six TCP connections per destination domain name and download objects in parallel over each of them.

The keep-alive mode in HTTP 1.1 is implicit. Both the client and server should support it by default, although it's not an error to not support it.

With HTTP/2, the HTTP protocol has been totally redesigned. Its primary goal is low latency with full request and response multiplexing. First, the *Connection* header is now forbidden and all clients and servers must persist connections. Second, a client can send multiple requests in parallel *on the same connection*, which is called *multiplexing*.

HTTP/2 introduced the notion of a stream, which allows a bidirectional flow of bytes to be exchanged over a persistent TCP connection. Each stream can carry one or more messages. A message is a complete sequence of frames that map to a logical request or response. Frames are the smallest unit in the architecture and each frame embeds headers or data. Read this article to learn more about the HTTP/2 protocol.

HAProxy support for keep-alive

HAProxy is a reverse proxy, also defined as a *Gateway* in the HTTP 1.1 specification. It has to connect clients to servers, even when each party speaks a different version of HTTP. HAProxy can allow an HTTP 1.1 client to communicate with an HTTP/2 server or vice-versa, for example. Whatever the scenario is, HAProxy will ease communication.

HAProxy supports persistent connections for the following versions of HTTP:

- Client side HTTP 1.1 since 1.4
- Server side HTTP 1.1 since 1.5
- Client side HTTP/2 since 1.8
- Server side HTTP/2 since 1.9

Reverse proxies stand between the clients and the servers and, of course, they have to follow the HTTP rules (RFC) for each version of the protocol. That said, since a reverse proxy collects the traffic of many users, it may not want to "mix" them over the same connection in some cases. Persistent connections between HAProxy and the server are flagged as "private" in the following cases:

- NTLM authentication is active between the client and the server. NTLM authenticates the TCP connection. So of course, this connection between the Proxy and the server can't be shared between multiple clients.
- Websocket: only the client who triggered the "upgrade" can speak over this connection.
- Transparent proxying: in this case, the proxy spoofs the client IP address to get connected to the server.

Even though HAProxy supports persistent connections on the server side, it will follow transparently and safely the principle of "private" connections for the cases described above.

There is one small drawback though: If there is a TLS SNI field to be sent to the server, defined by `sni <expression>` , the connection is marked as private, up to HAProxy 2.2.

HAProxy 2.3 improves this. It marks the connection as private only if the SNI is set using a variable expression. In the following HAProxy configuration example, the SNI field sent to the server comes from the variable expression *hdr(Host)*, which extracts the value from the HTTP *Host* header. This causes the connection to be marked private. However, it is not marked private when you use the `str` function, which sends a hardcoded string value:

view raw

```
backend servers
   # connections will be marked as private and can't be shared between users
   server srv1 10.0.0.1:443 check ssl sni hdr(Host)

   # not marked private, is reusable
   server srv2 10.0.0.2:443 check ssl sni str(my.domain.com)
```

There is a plan to provide connection pooling per `sni` in a future release of HAProxy, so that all connections with a variable SNI will not be marked as private anymore.

Keep-alive and server side connection pooling

As a reverse proxy, HAProxy has a strategic position in the architecture—between clients and servers. It benefits greatly from persistent connections in both directions. The cherry on top is that on the server side HAProxy can keep idle connections open for the next client, so there’s less wasted time and resources.

This behavior is managed by placing an http-reuse directive in a `backend` section, which can take the following values:

- *never*: all server side connections are considered “private” and HAProxy will never share a connection between multiple clients.
- *safe* (default): the first request of a client is always sent over a dedicated connection for the client. Subsequent requests from the client can be sent over existing idle connections.
- *aggressive*: the first request of a client can be sent over any connection that has been used at least once, providing the server properly supports persistent connections. Of course, a new connection may be opened in case no idle ones match the requirement described previously.
- *always*: the first request of a client is always sent on an existing idle connection. A new connection may be established if none of them are available.
  Note that it is recommended to use this mode only when the server delays closing the connection after sending a response to prevent a collision where HAProxy would send a request on a connection that’s being closed.

Benchmarking

Now, let's compare the different `http-reuse` modes within a very simple test lab:

- An injector, hey, simulates 50 users hammering the website for one minute.
- The backend server is just another HAProxy frontend which just returns "200 OK" empty responses. This server supports keep-alive, HTTP 1.1 and HTTP/2.

Note that in the tables below, HAProxy does not show its full potential because…

- HAProxy, the injector, and the server were all running on my small laptop
- IPTables and conntrack are enabled
- HAProxy was running in debug mode and streamedall info to a terminal's stdout
- Strace was attached to the HAProxy process

For these reasons, the performance number is totally useless as a raw value, but it can still be useful to compare the impact of persistent connections and the advantages of the HTTP/2 protocol framing model. I've marked the *Number of HTTP requests completed in 1 minute* column with an asterisk to indicate this.

Scenario 1: Connection close on the client side

In this scenario, I instructed `hey`, which acts as the client, to close the connection after each response. This is the worst case scenario!

I ran six tests in total, two for each of the three operating modes of `http-reuse`:

- HTTP 1.1 + safe mode
- HTTP 1.1 + aggressive mode
- HTTP 1.1 + always mode
- HTTP/2 + safe mode
- HTTP/2 + aggressive mode
- HTTP/2 + always mode

During each test, I collected the following information:

- The number of calls to the `connect()` syscall (using `strace -c`)

- The number of connections established on the server (using HAProxy's Runtime API command `show servers conn`)

This gives us the following results:

*** Results are limited by the test lab, see above**

HTTP Protocol version	HTTP reuse mode	Number of HTTP requests completed in 1 minute *	Number of calls to connect() syscall	Maximum connections used on the server side
HTTP 1.1	safe	22000	22000	50
HTTP 1.1	aggressive	22000	22000	50
HTTP 1.1	always	29000	54	50
HTTP/2	safe	20000	20000	50
HTTP/2	aggressive	30000	50	4
HTTP/2	always	30000	4	4

With both HTTP protocols, setting `http-reuse` to *always* provides much better results. That said, thanks to the streaming model of HTTP/2, *aggressive* mode provides good results as well with that protocol.

Scenario 2: Persistent connection on the client side

In this scenario, I instructed `hey` to keep the connections persistent for all 50 users. This is the best case scenario!

I ran six tests in total, two for each of the three operating mode of http-reuse:

- HTTP 1.1 + safe mode
- HTTP 1.1 + aggressive mode
- HTTP 1.1 + always mode
- HTTP/2 + safe mode
- HTTP/2 + aggressive mode
- HTTP/2 + always mode

During each test, I collected the following information:

- The number of calls to the `connect()` syscall (using `strace -c`)
- The number of connections established on the server (using HAProxy's Runtime API command `show servers conn`)

This gives us the following results:

*** Results are limited by the test lab, see above**

HTTP Protocol version	HTTP reuse mode	Number of HTTP requests completed in 1 minute *	Number of calls to connect() syscall	Maximum connections used on the server side
HTTP 1.1	safe	45000	59	50
HTTP 1.1	aggressive	53000	55	50
HTTP 1.1	always	55000	55	50
HTTP/2	safe	59000	51	50
HTTP/2	aggressive	62000	49	4
HTTP/2	always	78000	4	4

Conclusion on the tests

From these tests, we can draw these conclusions about HAProxy's `http-reuse` feature:

- When applicable, the *always* method hides the client side close mode to the server.
- The HTTP/2 streaming model reduces drastically the number of TCP connections required between HAProxy and the server.
- In all cases, setting `http-reuse` to *always* is best. So use it if your servers and environment allow it!
- *Safe* and *aggressive* modes have good results when the client can keep the connection persistent for several requests.
- The longer the distance between HAProxy and the server, the more benefit the reuse feature will bring to your global application response time.

Management of connection pooling

The magic behind the curtain of `http-reuse` is that HAProxy will manage a pool of TCP connections with the servers. Re-using already established connections allows you to get better performance from the network layer, hence the benefits for the application layer.

To manage the pools of connections, HAProxy provides multiple parameters that can be set on a `server` line in a `backend` section:

- *pool-low-conn*: When a thread needs a connection, it can take it from another thread unless the `pool-low-conn` threshold is reached for said thread. This ensures that each thread can keep some idle connections for itself and save the CPU cycles from this search. The default is 0, indicating that any idle connection can be used at any time.
- *pool-max-conn*: Set the maximum number of idle connections per server. 0 would mean no idle connections. HAProxy keeps these connections in a pool for later use with the next client request. The default is -1, which means “unlimited”.
- *pool-purge-delay*: Frequency at which HAProxy will close idle and unused connections. Only half of the idle connections will be closed after this period of time The default is 5 seconds.

Since HAProxy 2.2, there is a very useful command available in the HAProxy Runtime API: `show servers conn` . It provides live information about the number of connections and their state per server for each backend, as well as the number of idle connections per thread, and more. The information you get from this command will help you when setting the parameters seen above.

Keep-alive and server maxconn

The server directive’s `maxconn` parameter is a very well known feature of HAProxy. It can be used to protect servers against traffic spikes by routing requests to other servers or by queueing requests within HAProxy.

The server’s `maxconn` parameter was created at a time when HAProxy did not support keep-alive nor manage a connection pool (back in 2006, around HAProxy 1.2.X). At that point in time, one connection was equivalent to one request being processed by the server. So a `maxconn` of 100 meant that 100 requests at most could be processed in parallel by the server. By "limiting" the number of concurrent connections processing on the server, we could ensure that the server remained responsive, which was much faster than opening thousands of connections.

Nowadays, with the connection pooling model and, furthermore, with HTTP/2 streaming, the maxconn is not related anymore to the number of active connections, but instead applied to the number of active requests sent to a server. This means that with a `maxconn` of 100, a server could have:

- Fifty TCP connections established with HTTP/2, all of them being active, but a maximum of 100 HTTP/2 streams being sent over those connections.
- In HTTP 1.1, the number of connections per server should be equal to `maxconn`. Actually, to be very accurate, when you have a very fast server and a very small `maxconn` value, and you receive a spike in traffic, this can reach up to *maxconn + nbthread -1*. This is due to the pool algorithm "optimization" for performance. Managing this very rare case would cost useless CPU cycles for all the other cases…Furthermore, this combination is unlikely to happen ☺

Conclusion

In this blog post, you learned that HAProxy supports persistent connections in its various forms depending on the version of HTTP. While HTTP 1.0 and 1.1 made use of the *Connection* header to enable or disable persistence, HTTP/2 always expects persistent connections. HTTP/1.1 introduced the concept of pipelining to send multiple requests over a single connection in sequence, but HTTP/2 gave us multiplexing, which allows browsers to send multiple requests in parallel over a single connection.

HAProxy leverages persistent connections on the server side to support connection pools, wherein idle connections to backend servers can be reused among clients. You can control the behavior of the connection pool by setting the `http-reuse` directive in a `backend` section. Benchmarks indicate that setting it to *always* is typically best. Fine tune it more with the `pool-low-conn`, `pool-max-conn`, and `pool-purse-delay` parameters on a `server` line.

Finally, you learned that the `maxconn` setting has evolved over time. In the beginning, it controlled how many connections to establish with a server. Nowadays, it indicates how many requests to send at once. With connections being reused, this fits better with the amount of work you're truly sending.

SHARE THIS ARTICLE

Tags: HTTP Keep-Alive, persistence

1 Comment

Dmitriy Alekseev on December 16, 2020 at 3:01 am

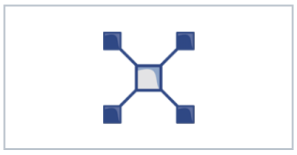
Great article, thanks you

Reply

This site uses Akismet to reduce spam. Learn how your comment data is processed.

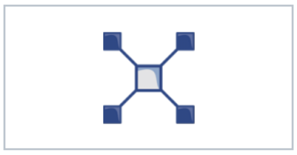
Comments will display after being approved by the moderator.

RELATED STORIES



Microsoft Remote Desktop Services (RDS) Load-Balancing

HAProxy and HTTP Errors 408 in Chrome



Client IP Persistence or Source IP Hash Load Balancing

Connect With Us

+1 (844) 222-4340

contact@haproxy.com

Solutions

- Load Balancing
- High Availability
- Application Acceleration
- Security
- Administration
- Microservices
- Kubernetes
- Web Application Firewall

Products

- HAProxy Enterprise
- HAProxy Kubernetes Ingress Controller
- HAProxy ALOHA
- HAProxy Edge
- HAProxy Fusion Control Plane
- HAProxy One

Support

- Customer Support Portal

Resources

- HAProxy Enterprise Documentation