# HTTP/1.1 Keep Alive

HyperText Transfer Protocol



In this segment, I'm going to cover one very important optimization that occurred in HTTP/1.1, something called the Keep Alive header.

# HTTP/1.0

- Open connection
- Issue GET
- Server closes connection after response



HTTP is a basic request, response protocol. HTTP/1.0 was very simple. A client wanting to request a document opens a connection. It sends a GET request. The server responds with a status code, such as 200 OK, the document, and closes the connection once the response is complete. If the client wants to request a second document, it must open a second connection.

When the web was mostly text, with maybe an image or two, this approach worked just fine. People hand-wrote their web pages, putting in all of the formatting.

# HTTP/1.0 speed

- Latency: 50ms
- Request size: 1 full segment
- Response size: 2 full segments (size of slow start window)
- Segment packetization delay: 10ms (request and response), full duplex
- Maximum open connections: 4

- Case 1: Single page, 230ms
  - SYN: 50ms, SYN/ACK: 50ms, ACK/request: 60ms, response: 70ms
- Case 2: Page that loads 2 images: 480ms
  - Setup: 100ms, request/response: 130ms
  - Setup: 100ms, request/response: 150ms



Recall our results from analyzing HTTP/1.0. Loading a single page takes 230 milliseconds, and loading a page with two images takes over twice as long. A lot of this time is spent opening connections, and if we could request more documents at once it could be much faster.

# HTTP/1.0

- Open connection
- Issue GET
- Server closes connection after response
- *Opening many connections is slow*
- *Many transfers are small, doesn't let TCP window grow*



So the approach that HTTP/1.0 uses can be really wasteful. Clients spend a lot of time opening connections. Furthermore, the TCP congestion control window doesn't get a chance to grow, since each connection has a new window.

# HTTP/1.1

- Added Connection header for requests
  - keep-alive: tells the server "please keep this connection open, I'll request more"
  - close: tells the server to close the connection
  - Server can always ignore
- Added Connection header for responses
  - keep-alive: tells the client it'll keep the connection open
  - close: tells the client it's closing the connection
- Added Keep-Alive header for responses
  - Tells client how long the connection may be kept open



HTTP solved this problem by adding a few headers to requests and responses. A request can include a Connection header, which tells the server whether it would like the connection to be kept open after the response or closed. The server can do whatever it wants, but the client can give a hint. For example, if you're requesting a basic text file, there's no reason to keep the connection open, as the text file won't reference other things to load.

A response includes a Connection header, which tells the client what the server decided to do. If it decided to keep-alive the connection, then the keep-alive header tells the client for how long. Now, the client can send further requests on the same connection. It can also open more connections, if it wants, but it doesn't have to.

# HTTP/1.1 speed

- Latency: 50ms
- Request size: 1 full segment
- Response size: 2 full segments (slow start window is 30 segments)
- Segment packetization delay: 1ms (request and response)
- Maximum open connections: 2
- Page that loads 11 images

- HTTP/1.0 speed: 1,421ms
  - Page setup: 100ms, request/response: 103ms
  - 11 images. 6 x (image setup: 100ms + request/response: 103ms)
- HTTP/1.1 speed: 326ms
  - Connection setup: 100ms
  - Page request/response: 103ms
  - Image requests/responses: 123ms



So it turns out this is a big deal. Let's consider a more realistic case than before, where the packetization delay is only 1 millisecond and the page loads 11 images. Browsers today usually have more than 2 open connections, but they also load more than 11 resources, we'll just keep these numbers small for simplicity. We're going to use the same analysis we used when looking at HTTP/1.0 in the HTTP/1.0 video. The slow start window is big enough that we'll never hit congestion control.

For HTTP/1.0, this will take 1,421 milliseconds. There are seven rounds. In the first round, we request a page. This takes 203 milliseconds. In the next 6 rounds, we request 2 images each, except for the last round, when we request one image. Each round takes 203 milliseconds. So the total time is 203 milliseconds plus 1218 milliseconds, for 1.421 seconds.

For HTTP/1.1, this will take 326 milliseconds! We setup the connection, that takes 100 milliseconds. Requesting the page takes another 103 milliseconds. Requesting the 11 images, however, only takes 123 milliseconds! That's 51 milliseconds for the first request, and 72 milliseconds for the 11 responses, 50 milliseconds of latency plus 22 milliseconds of packetization delay. It's almost four times faster, because we can send these requests back-to-back in a single connection and don't have to open new connections.

# SPDY

- Protocol proposed by Google to speed up the web
- Request pipelining
- Removes redundant headers
- Becoming basis of HTTP/2.0



HTTP/1.1 has been around for a while, since 1996 or so. Very recently, Google has developed a new protocol, called SPDY, that improves on HTTP. It does things like allow request pipelining. One issue HTTP sometimes runs into is that the order in which a client requests resources is the same that the server responds. This can be a problem if some resources require a lot of processing. Say you have a dynamically generated web page through something like Ruby on Rails or Django. Your database is overloaded, so it takes a while to generate the page. But most of the resources are just images that can be send quickly. If the client requested the slow page first, it won't receive any of the images until after it receives the page. It would be nice if the server could respond in a different order, and say start sending the images while the page is being generated.

SPDY also removes redundant headers. Open up Wireshark and look at some HTTP requests and responses. Very often, there's a lot of redundant information in each response and request. If you could just set some parameters (such as browser type) for the duration of the session, rather than send it each time, that would speed things up.

SPDY has been in use for a while, and it's becoming the basis of HTTP/2.0. In a few years, I suspect most sites will be using HTTP/2.0 because of the speed benefits it will bring, especially for mobile devices.