EDI: Third Lab Report

Aiman Al Masoud - 502044

June 10, 2022

Abstract

1 Parallel Connections

Page Load Time (PLT) is one of the most critical performance evaluation metrics for a website, as it correlates directly to user experience.

PLT takes into account both the time to download the html of a page, as well as the time to download all of its embedded objects, as a browser would typically do when directed to navigate to a page.

An acceptable Page Load Time for a website in 2022 ranges between 1 and 2 seconds, but it depends on the context, and having an even lower Page Load Time is desirable; users obviously tend to abandon websites that take more time to load than their competitors. [**??**]

1.1 Firefox's About Config

The first experiment that was performed, pertains to the effect of parallel TCP connections on Page Load Time. As power users will know, modern web browsers allow them to peek into various configuration settings and tweak them, this is especially true of Mozilla Firefox, where the user can access various technical settings by typing in 'about:config' into the navigation bar, and hitting enter.

More specifically, one of these configuration parameters is called 'network.http.max-persistent-connections-per-server', and, as the name suggests, it governs the maximum amount of concurrent persistent TCP connections that the browser is allowed to sustain at a time, with any particular web server over the Internet. [**??**]

By default, the value for this option is set to '6', but it can be changed. Does decreasing it or increasing it have an effect on PLT? Do higher values of this parameter lead to performance improvements when loading pages?

1.2 HTTP/1 vs HTTP/2

Here comes to play an important distinction between websites, and that is difference between websites running on HTTP/1 and HTTP/2: the first two major versions of the ubiquitous Hypertext Transfer Protocol.

There are various ways in which HTTP/2 tried to improve on HTTP/1, but what is of specific interest to us in this particular inquiry is that: websites running on HTTP/1 tend to favor connections to multiple separate servers, to download the objects of a page in parallel, and overcome the limitation imposed by browsers on the number of parallel connections to a single server, a technique known as Domain Sharding [**??**].

HTTP/2 helped overcome this very necessity, by introducing Streams that could convey multiple concurrent (and bi-directional) HTTP requests within a single persistent TCP connection; thus eliminating the need to connect to separate resource servers, and even eliminating the overhead associated to having to open multiple TCP connections with them. This is something that, at least on the long run, will render the technique of Domain Sharding obsolete [**??**].

So, to go back to the original question: resource intensive HTTP/1 websites may benefit from an increased number of parallel TCP connections, as they only support a single HTTP request per TCP connection. On the other hand, HTTP/2 websites support multiple concurrent HTTP requests over the same TCP connection, hence, from a theoritical viewpoint, it shouldn't make much of a difference to them if the client allows them to open multiple TCP connections.

1.3 Methodology and experimental setup

The experiment was performed in practice, by measuring the PLTs of 6 different websites (3 HTTP/1 and 3 HTTP/2 websites) under two different conditions: 1 and 6 maximum connections per server.

The browser that was employed for the experiment was Firefox, and the number of maximum connections was set tweaking the aforementioned: "network.http.max-persistent-connections-per-server" parameter.

Although the experiment summarized in table 1 isn't exhaustive, it clearly shows that HTTP/1 websites fare much better when granted a higher number of connections per sever.

		Average PLT (seconds)
HTTP Version	**Connections**	
1	**1**	3.017778
	6	2.212222
2	**1**	3.178889
	6	3.486667

Table 1: More TCP connections improve the performance of HTTP/1 websites, not HTTP/2 ones.

2 Caching Policies

Cache is crucial to the correct operation web infrastructure, as its valid usage provides advantages for the origin servers, as well as for the end users of a particular web-service.

From the service provider's point of view, cache helps to optimize the number of accesses made by clients to the origin web server, easing up the load on them;

From the client's side, cache helps in reducing PLT, especially if the assets (images, stylesheets, scripts) that have to be served with the page are a lot, and don't change too often.

2.1 Remote vs Local

Caches can be remote or local. Remote caches can either fall under the category of Proxy or Managed caches. Proxy caches follow a ”pull model”, where any client that accesses content X from a particular local network, will trigger the caching of content X on the local network's proxy, so as to avoid having to download the content again from the wider Internet, when another client within the same network requests it. Managed caches follow a ”push model”, in that the service providers manually load content to the cache, with the goal of easing up the load on their origin servers, by distributing client requests over multiple servers. An example of a Managed cache is a Content Delivery Network (CDN).

2.2 Public vs Private

Caches can also be public or private, the idea behind private caches is to avoid disclosing private information that has to be cached on the user's side.

All of these caching policies and behaviors, and many more complex ones, are governed by a set of specific HTTP request and response headers, such as the Cache-Control, If-Modified-Since and ETag headers. [**??**]

2.3 Methodology and experimental setup

The goal of this experiment was to assess whether the caching policies of various websites proved to be effective, at least on the client's side, by comparing the performances of said websites with and without cache enabled.

This was done on a real browser (Firefox), with the help of a custom bash script with the xdotool [**??**] utility, to automate the process of reloading the web-pages, and collecting the data consistently, taking several readings. The script in question, dubbed ”scripton.sh”, is available on the Github repository associated to this report [**??**], but it has to be fine tuned to the screen-dimensions of the particular host computer, as it leverages the position of the cursor on the screen to interact with Firefox and copy the textual output from the Network Monitor's interface; this process of fine-tuning can be performed manually, with the help of the included ”calibrate.sh” script.

Chosen for this test were 9 websites (3 secure HTTP/1, 3 unsecure HTTP/1 and 3 HTTP/2), and the data collected for each was:

1. the PLT, and
2. the number of requests made to the server

As it is evident from the results in table 2, in all cases, caching does prove to be advantageous for the client, as it saves quite a bit of time loading the page, decreasing the number of requests that have to be performed towards the server.

		number_requests	**plt_seconds**
http_version	**cache**		
http1	**False**	39.950000	2.343500
	True	29.923077	1.543077
http2	**False**	80.083333	3.708333
	True	56.333333	2.652667
unsecure-http1	**False**	78.666667	4.010000
	True	45.500000	3.580000

Table 2: Average results with and without caching

3 Apache Benchmark

There are several automated CLI benchmarking tools to measure the performance of webservers, and Apache Benchmark (AB) is one of them. Unfortunately, AB doesn't support the second version of the HTTP protocol, but it provides several options to test the performance of HTTP/1 servers, and especially Apache webservers.

3.1 Methodology and experimental setup

The idea behind this experiment was to test the effects of the concurrency (-c) and keep-alive (-k) options of Apache Benchmark, on the performance of 3 websites, when issuing a fixed amount of requests (20).

The concurrency (-c) option regulates the number of multiple requests to perform at a time; since the servers in questions are HTTP/1 only servers, concurrent requests will have to be performed over multiple TCP connections.

The keep-alive option (-k) specifies whether to request the usage of the HTTP KeepAlive feature (aka: persistent connections), which means that a single TCP connection will be re-used for multiple sequential HTTP requests, thus reducing the overhead of TCP connection creation [**??**]. Obviously, since this is HTTP/1, concurrent requests cannot share the same TCP connection, as already stated.

Each website was tested with the following 4 commands:

1. ab -c 1 -n 20 http://example.com/

   20 requests, one at a time, without reusing the TCP connection.

2. ab -c 10 -n 20 http://example.com/

   20 requests, 10 at a time, without reusing TCP connections.

3. ab -k -n 20 http://example.com/

   20 requests, one at a time, reusing the TCP connection.

4. ab -c 10 -k -n 20 http://example.com/

   20 requests, 10 at a time, reusing TCP connections.

The first command (-c 1 and no -k) is expected to be the slowest, as the requests are all performed sequentially, and, for each request, a new TCP connection must be opened, with all of the overhead associated to that costly operation.

The last command instead (-c 10 and k), is expected to be the fastest, as it combines the benefits from issuing multiple parallel requests, with the benefits of reusing already opened TCP connections.

Sure enough, these are the average times that it took to run these tests:

Options	Average Time (seconds)
-c1	5.191
-c1 && -k	3.194
-c10	1.036
-c10 && -k	0.939

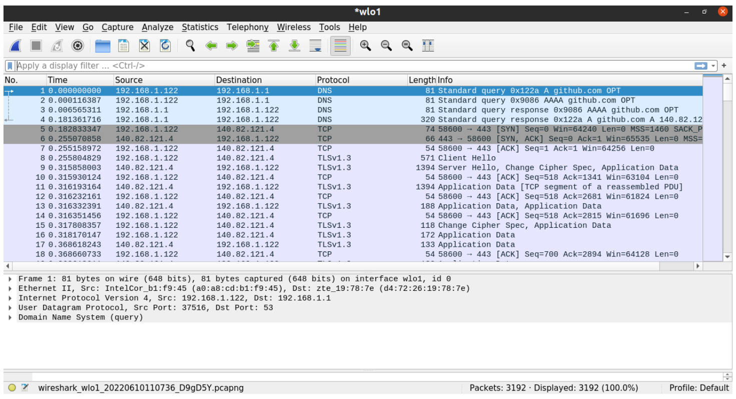
Table 3: Average times to run tests with fixed workload

4 nghttp and h2load

Among the tools that can be used to measure the performance of HTTP/2 servers are nghttp, which is an HTTP/2 client, and h2load, which is a benchmarking tool.

4.1 The Role of Warmup Time in h2load

Warmup time is used by the benchmarking tool's client to start a connection with the webserver, before actually beginning to transfer benchmark data [**??**], you can verify that some traffic is exchanged during the warmup phase using a sniffer such as Wireshark, as figure 1 shows:


*wlo1
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time         Source         Destination    Protocol  Length Info
   1 0.000000000  192.168.1.122  192.168.1.1    DNS          81 Standard query 0x122a A github.com OPT
   2 0.000116387  192.168.1.122  192.168.1.1    DNS          81 Standard query 0x9086 AAAA github.com OPT
   3 0.006565311  192.168.1.1    192.168.1.122  DNS          81 Standard query response 0x9086 AAAA github.com OPT
   4 0.181361716  192.168.1.1    192.168.1.122  DNS         320 Standard query response 0x122a A github.com A 140.82.12
   5 0.182833347  192.168.1.122  140.82.121.4   TCP          74 58600 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_P
   6 0.255070858  140.82.121.4   192.168.1.122  TCP          66 443 → 58600 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=
   7 0.255158972  192.168.1.122  140.82.121.4   TCP          54 58600 → 443 [ACK] Seq=1 Ack=1 Win=64256 Len=0
   8 0.255804829  192.168.1.122  140.82.121.4   TLSv1.3     571 Client Hello
   9 0.315858003  140.82.121.4   192.168.1.122  TLSv1.3    1394 Server Hello, Change Cipher Spec, Application Data
  10 0.315930124  192.168.1.122  140.82.121.4   TCP          54 58600 → 443 [ACK] Seq=518 Ack=1341 Win=63104 Len=0
  11 0.316193164  140.82.121.4   192.168.1.122  TLSv1.3    1394 Application Data [TCP segment of a reassembled PDU]
  12 0.316232161  192.168.1.122  140.82.121.4   TCP          54 58600 → 443 [ACK] Seq=518 Ack=2681 Win=61824 Len=0
  13 0.316332391  140.82.121.4   192.168.1.122  TLSv1.3     188 Application Data, Application Data
  14 0.316351456  192.168.1.122  140.82.121.4   TCP          54 58600 → 443 [ACK] Seq=518 Ack=2815 Win=61696 Len=0
  15 0.317808357  192.168.1.122  140.82.121.4   TLSv1.3     118 Change Cipher Spec, Application Data
  16 0.318170147  192.168.1.122  140.82.121.4   TLSv1.3     172 Application Data
  17 0.368618243  140.82.121.4   192.168.1.122  TLSv1.3     133 Application Data
  18 0.368660733  192.168.1.122  140.82.121.4   TCP          54 58600 → 443 [ACK] Seq=700 Ack=2894 Win=64128 Len=0

Frame 1: 81 bytes on wire (648 bits), 81 bytes captured (648 bits) on interface wlo1, id 0
Ethernet II, Src: IntelCor_b1:f9:45 (a0:a8:cd:b1:f9:45), Dst: zte_19:78:7e (d4:72:26:19:78:7e)
Internet Protocol Version 4, Src: 192.168.1.122, Dst: 192.168.1.1
User Datagram Protocol, Src Port: 37516, Dst Port: 53
Domain Name System (query)

wireshark_wlo1_20220610110736_D9gD5Y.pcapng    Packets: 3192 · Displayed: 3192 (100.0%)    Profile: Default
```

Figure 1: captured during warmup time

The packets were captured during the warmup phase of the command, which specifies 1 second of warmup time and 5 seconds for the duration of the actual benchmark:

h2load –warm-up-time 1 https://github.com –duration 5

When warmup time is greater than 0, "time for connect" and "time to 1st byte" tend to become 0, as the following tables show:

	min	max	mean	sd	+/- sd
time for request:	0us	0us	0us	0us	0.00%
time for connect:	157.24ms	157.24ms	157.24ms	0us	100.00%
time to 1st byte:	227.56ms	227.56ms	227.56ms	0us	100.00%
req/s	0.00	0.00	0.00	0.00	100.00%

Table 4: Without warmup time (warmup time = 0)

	min	max	mean	sd	+/- sd
time for request:	106.22ms	164.90ms	134.89ms	14.35ms	67.57%
time for connect:	0us	0us	0us	0us	0.00%
time to 1st byte:	0us	0us	0us	0us	0.00%
req/s	7.40	7.40	7.40	0.00	100.00%

Table 5: With warmup time = 1 second

4.2 Embedded Objects with nghttp

Checking the Page Load Time of a website usually implies testing it out on a real browser; but nghttp, being a fully featured HTTP/2 client, supports downloading the embedded objects/assets of a web page, besides its html code, through the -a (–get-assets) option [**??**].

This means that it is possible to use nghttp to approximate the time that it would take for a real web browser to download the assets of a page from a web server.

This kind of test has been carried out for 3 HTTP/2 pages, and the dumps are available on the repository. [**??**]

As an example, the following tables represent the behaviors of one of those websites (https://archive.org) when downloaded using nghttp, with and without the -a flag:

id	responseEnd	requestStart	process	code	size	request path
13	+308.00ms	+175us	307.82ms	200	1K	/

Table 6: without the -a option

id	responseEnd	requestStart	process	code	size	request path
13	+408.96ms	+151us	408.81ms	200	1K	/
15	+717.76ms	+409.29ms	308.47ms	200	318K	/offshoot_assets/index.56c7d2ac8e12.css
17	+717.97ms	+409.30ms	308.67ms	200	1K	/offshoot_assets/vendor/lit@2.0.2/polyfill-support.js
19	+718.15ms	+409.30ms	308.85ms	200	1K	/offshoot_assets/vendor/@webcomponents/webcomponentsjs@2.6.0/webcomponents-loader.js
21	+718.31ms	+409.30ms	309.01ms	200	2K	/offshoot_assets/js/webpack-runtime.278ae0d32b8bdea6e95a.js
23	+1.95s	+409.30ms	1.54s	200	103K	/offshoot_assets/js/index.2d907721752dd736f0e5.js

Table 7: with the -a option

As one can see from the tables, the -a option can give a rough idea of the PLT of the page on an actual browser, keeping in mind that on an actual browser the ”perceived” PLT will always be higher, as this kind of test doesn’t take into account the time needed for the web engine to render the content and run the scripts.

5 Conclusions

The source code of this report, as well as that of the tests, and the data that was collected, is all available on the associated Github repository [**??**]. The data and scripts for each of the four sections of this report are organized in 4 different subfolders, within the /lab3/expriments/ directory[**??**].

6 References

1. `https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/`
2. `https://www.computerworld.com/article/2541429/hacking-firefox--the-secrets-of-about` `html?page=5`
3. `https://blog.stackpath.com/glossary-domain-sharding/`
4. `https://httpd.apache.org/docs/2.4/programs/ab.html#synopsis`
5. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching`
6. `http://manpages.ubuntu.com/manpages/trusty/man1/xdotool.1.html`
7. `https://github.com/aiman-al-masoud/edi_reports`
8. `https://nghttp2.org/documentation/h2load.1.html`
9. `https://nghttp2.org/documentation/nghttp.1.html`