HTTP/2 Optimization

Spencer Fricke, Christian Krueger, and Emmanuel Contreras Guzman

Abstract— **The HTTP/1.1 (h1.1) protocol has been around since 1996, and is still in used today. Although it is currently the main protocol used to transfer data in web browsers, it contains many disadvantages and limitations that will hinder the performance of future networking practices. As website and web apps in the near future become more demanding, we will see an impact in network performance in which the protocol will become the bottleneck. Many of these limitations can be resolved by adopting the new HTTP/2 (h2) protocol. Although readily available, its adoption has been slow. This has been due to a lack of motivation on the developer's part to see the benefits of adopting h2 vs h1.1 - cost-to-improvement ratio. Because h2 is backwards compatible with h1.1, developers can continue to implement their well known workarounds without utilizing h2's improvements. Another point for its slow adoption is the lack of best practices to optimize websites and applications. Developers do not have a clear set of rules to best harness h2's improvements.**

Although in our research we were not able to determine the optimum number of files, file sizes, size transfer structure, and server environment that would make using h2 advantageous for developers, we were able to create general guidelines for using h2. In our research, we also encountered a few unforeseeable issues we had to resolved which we hope will save time for future researchers in this field. Lastly, we have made all the code and documentation available for others to use and replicate our experiment on the links below.

https://github.com/sjfricke/HTTP2-Optimization-Research
https://http2optimization.com/

Keywords— **http optimization h2 h1.1 server push**

I. Introduction

With the extremely fast paced evolution and adoption of gadgets, devices and computers, one would think that protocols would evolve and integrate into society just as fast. Unfortunately this is not the case. The majority of the internet is still using the h1.1 protocol released in 1996 to send and receive data to each computer's browser. Although this protocol has proven to be very reliable and effective for our current application demands, it does have disadvantages and limitations which we will need to overcome in the near future as websites and web apps become more demanding and complex. h1.1 had a successor protocol developed by Google called SPDY, which although solved some of h1.1's problems, was never widely adopted [2]. After seeing the advantages of the experimental protocol SPDY, Hypertext Transfer Protocol working group set out to create the h2 standard based on it. The development team for h2 included some of the developers of SPDY. h2 includes many improvements and optimizations introduced in SPDY, and is the foreseeable replacement of h1.1. Unfortunately, its adoption has been slow. This has been due to two main points: 1) slow developer adoption due to lack of cost-to-improvement ratio, and 2) lack of best practices and principles developers can adopt to best utilize h2 [5].

In order to facilitate the transition, h2 has been implemented to be backwards compatible with h1.1. If either the client or server does not support h2, http transmission will fall back to h1.1. This has provided a lack of incentives to developers to trade their long used h1.1 "hacks" for time invested learning how to best use h2. Some of these "hacks" include: concatenating and minify javascript files into a single file before sending, spriting images and domain sharding [4].

Although these hacks improve the performance of h1.1, they each come with their own drawbacks. Concatenating and minifying files defeat the purpose of caching files on a client's computer as the whole file needs to re-download with any minute change in the code. Furthermore, performance gain from bundling the files together is not seen. Spriting images, although it reduces the header overhead by sending one file, encounters the same caching problem, changing one image in the sprite requires the whole sprite to be re-downloaded. Finally, domain sharding introduces many fault points in transmission that could increase transmission time as the files are spread throughout different servers.

Because of h2 backwards compatibility, and because h1.1 is currently "fast enough" for developers needs, there is no incentive for developers to adopt this new standard. Wider adoption will be seen when transmission speed becomes a bottleneck for developers websites and web apps, and the only way to improve the speed is to adopt h2.

The lack of incentive to adopt h2 instead of current “hacks” has also impeded the creation of a clear set of rules for developers to follow. Without developers using h2, there are not many suggestions by the community on best practices to adopt.

In our research, we aimed at solving the second point slowing down the adoption of h2 by creating a set of best practice guidelines for developers. We did this by analysing the performance of different automatically generated websites and changing one variable at a time in order to determine best transfer performance. For example, when creating a website containing 1 MB worth of javascript files, currently it is unclear if when using h2, we would get the best performance by sending one file at 1MB, 1024 files at 1 KB, or somewhere in between. If the answer is somewhere in between, we aimed to determine if it was a consistent ratio we could extrapolate to optimize future websites. We also planned to find the performance achieved by the structure the HTML file requests its assets, the server framework used (Nginx vs Apache) and the amount of latency available (Ethernet vs Wifi).

II. Materials

A. Hardware

For the client-host interaction, we began the project using a raspberry pi 3 to host our public web servers, a generic household router and both a high-end desktop computer and another raspberry pi 3 for loading the generated sites. Due to issues with raspberry pi’s performance, mentioned in detail in the discussion section, we switched over to Amazon Web Services (AWS), as well as used laptop with an intel dual core processor to gather our data. We used Charter Spectrum ISP for this project, which has a maximum capable transfer rate of 60 Gbps.

B. Software

All of the raspberry pi’s, laptop and desktop ran the Ubuntu 16.04.2 LTS Linux distribution. The servers (AWS, Raspberry Pi’s, Laptop) had Nginx 1.10.0 web servers or Apache 2.4.25. A mysql database was used to store all of the data from the HTTP Archive format file (HAR). BASH and python were used to procedurally generate the test websites. Node.js was used to retrieve, parse, store and analyse the website’s metrics. Headless Chrome and Firefox were called by the scripts to mimic GET requests.

III. Website Generation

Website generation was provided by two custom bash scripts. One to procedurally generate the websites given certain parameters, and a second master-script to call the generation script and provide an iterative list of parameters. The result is an array of generated websites all with different sizes, object count, ordering method and file type, see Table I. Each website resides in a directory whose name reflects its contents ie different parameters. Each website’s metadata also reflects its contents for easier parsing. The bulk of the generation is done via the linux command dd.

Each object in the sites were all filled with garbage data, ie dd uses an input file /dev/random. This is to simulate actual metadata while being able to fill the objects to the desired file size. The data can be random due to us only concerned about gathering of the files as the speed loading the data is independent of the speed to transfer the data to the client.

TABLE I
Variations of websites generated by the Website Generator Script

	WXYZ/abcd																				
	100k																				
	250k																				
	500k																				
	750k																				
File Size	1M																				
	1.5M																				
	2M																				
	2.5M																				
	4M																				
	6M																				
	8M																				
		1	2	3	4	5	6	7	8	9	10	30	35	50	70	90	100	125	150	175	200
		Objects																			

IV. Website Requests

A. Capture of HAR Files

When an HTTP request is made, many of the browser-server interactions are logged in an HTTP Archive format file with the extension ".har". This har file is a JSON-formatted archive file that contains logs of many metrics captured during the browser-server interaction. The file logs load times, file sizes, server hosting the site, browser name and version to name a few fields. With this file in mind, we created a script using JavaScript (HAR_Headless_Parser.js) that would make a request to each one of the websites we generated .

When running the script, we passed in database information as well as a list of the generated websites. The script had three main functions: 1) request the website using chrome headless, 2) parse fields of interest from the captured HAR file, and 3) insert extracted fields into SQL database.

When trying to capture the HAR files from the chrome headless browser, we ran into an issue due to the raspberry pi not having a display to output to. We overcame this problem by using a program called X video framebuffer (Xvfb) [3]. This program simulates a graphical screen in a terminal so the chrome browser has a place to output graphical memory buffer data to.

B. Parsing HAR Files

The second function of the HAR_Headless_Parser.js script was to parse out the fields we deemed necessary in order to run our analysis and compare each website's performance. Because the format of the har file was in JSON, which contains its data in arrays or dictionaries, it was relatively simple to extract the data by either using key-value pairs, or iterating through arrays to find the values. Once the values were found, they were placed in temporary variables later used for inserting into the database.

C. Inserting Into SQL Database

Before running the HAR_Headless_Parser.js script, we prepared an SQL database hosted on the desktop computer used for main control. We created two tables: Website, and Entries. The table "Website" contained ten fields and was used to store information about the main request being made. The table "Entries" contained twenty nine fields, and was used to store information about the different files that made up the website.

The last function of the HAR_Headless_Parser.js script was to insert the fields parsed into the SQL database. This was done using the JavaScript mysql connect command. We prepared two queries that inserted the data from the temporary variables into the database. Table II lists the different fields being parsed out of the har file and inserted into the database.

TABLE II
List of tables and fields stored in the database

Websites Table	**Entries Table:**	
1. WebsiteID	1. Entry ID	16. ResponseStatus
2. Domain	2. Website ID	17. ResponseHeadersSize
3. NumberOfFiles	3. Domain	18. ResponseBodySize
4. StartedDateTime	4. StartedDateTime	19. ResponseHttpVersion
5. OnContentLoad	5. TotalTime	20. ResponseTransferSize
6. OnLoad	6. RequestCacheControl	21. Blocked
7. ObjectType	7. RequestDate	22. DNS
8. Size	8. RequestUserAgent	23. Connect
9. Count	9. RequestHeadersSize	24 Send
10. Structure	10. RequestBodySize	25. Wait
	11. RequestUrl	26. Receive
	12. ResponseDate	27. SSLTime
	13. ResponseLastModified	28. ComputerType
	14. ResponseServer	29. ConnectionPath
	15. ResponseContentLength	

D. Cleaning The Data

During our analysis, we observed that the data contained some large outliers in transfer times, possibly due to network delays or flow control. We generated a script (outlier-finder.js) that detected outliers and removed them from our dataset. This script worked by keeping a running sum of the transfer times of previous files, if a transfer time exceeded ¾ of the running average, it was excluded from the analysis dataset. Because this script removed less than 10% of the data points from any one data point, the results from our analysis still remain valid. Overall, we observed smoother, more straight curves after cleaning up the data from these outliers.

E. Validation

In order to be certain that the data we collected was accurate and not random, as well as to observe that it was in fact taking advantage of h2's features, we decided to run a set of the same data using h1.1. These sets were all ran using nginx on AWS.

V. Analysis

Using our HAR_Headless_Parser.js script we generated data using AWS and the laptop for an estimated 30 hours, collected 3,000,000 rows of data and transfer a bandwidth totaling ~500GB. Making sense of this data was a difficult task. In order to best visualize the millions of data entries, we used javascript to generate google charts which plotted transfer time vs the other variables in order to observe any trends.

A. Server Push

We tested server push by transferring the different set of generated websites with server push enable and disabled when using an Apache server. When searching for server push on nginx, we realized that there was no server push in their h2 implementation and there won't be in the near future. According to Mozilla developers, it is a feature that not many web developers are using and therefore they are focusing their energy elsewhere.

B. Effect of Structure, File Sizes and Number of Files on Application Transfer Times

In order to analyze the performance of the websites with varying parameters, we set up two instances of AWS acting as a server-client and had them run the HAR_Headles_Parser.js continuously transferring generated websites many times in order to average the transfer times to account for variability.

C. Wired vs Wireless Performance of h2

In order to test the performance of h2 over a wired and wireless connection, we set up the laptop and desktop requesting the same websites from AWS. The desktop had a wired connection and the laptop had the wireless connection.

VI. Main Results

Due to the immense variation of website, these results are limited to our specific website design. We do think that some of these results can be extrapolated to other website structures. If developers are using these findings, they should evaluate how much their website structure resembles ours.

A. Validation

As you can see from Figure 1, the data collected of h1.1 vs h2 for the same set of websites shows that h2 has significantly better performance across all servers, except when transferring between two to ten files. h2 was about a quarter second slower in this range likely due to h1.1's six parallel TCP connections. We also observed that h2 is significantly faster when transferring one file across all websites we tested. We believe this is due to header compression.

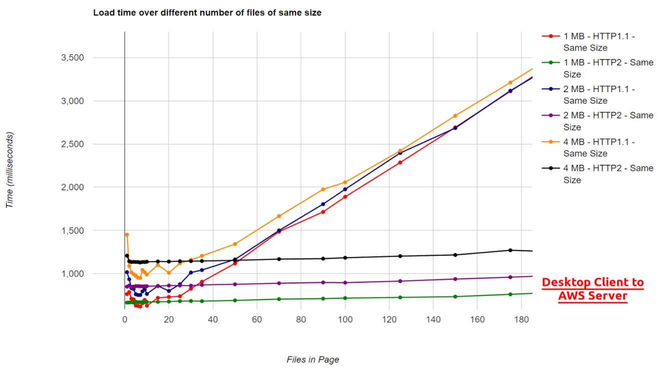


Figure 1. h1.1 vs h2 performance comparison table

B. File Size and Number of Files

Looking at File Size and Number of Files, we determined that if a website has less than fifty files, it is still faster to concatenate, but h2 scales with more files so performance will be about same with or without concatenation.

Figure 2 shows that after fifty files, there is half a second increase in delay when compared to the initial less than ten file average. This translates to 72% increase in delay.

Figure 3 shows that h2 is still faster with concatenation, but h2 scales much better when there are more files. The performance increase of ~3.3% does not justify the loss of caching, which would have a larger impact in performance overall.

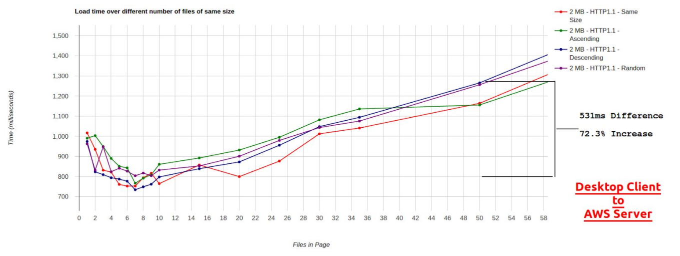


FIGURE 2. SAME SIZE WEBSITES OVER H1.1

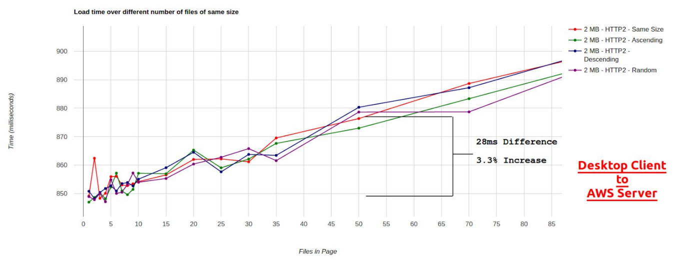


FIGURE 3. SAME SIZE HTTP2 - LESS FILES

If you have very large website with two hundred or more files, h2 doesn't scale if hardware is the bottleneck. Figure 4 show how at around five hundred files, the raspberry pi transfer time increases at a higher rate, likely due to the processor's inability to handle the amount of incoming packets.

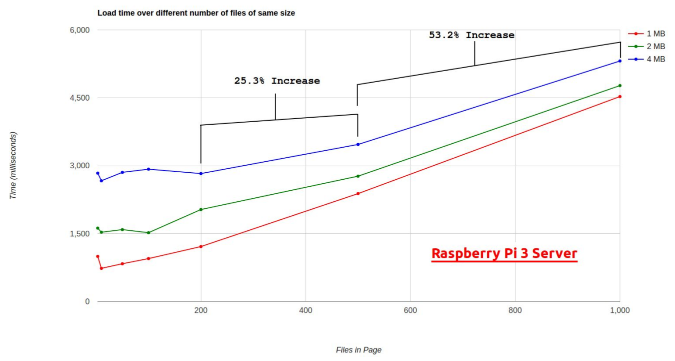


FIGURE 4. RPI PROCESSING BECOMES A BOTTLENECK AT ~500 FILES.

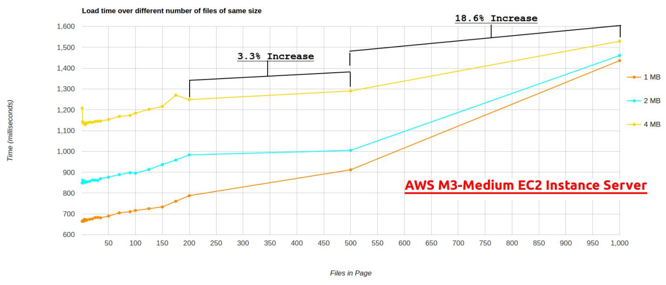


FIGURE 5. H2 SCALES WELL IF HARDWARE IS NOT THE BOTTLENECK

Figure 5 shows the same set of data but with AWS. As you can see, we originally thought that h2 didn't scale after about 500 files, but with further testing we found that the Rpi was the bottleneck, h2 does indeed scale as the number of files on a website increase. Developers should make sure that the server in which they are hosting their application can handle the traffic to avoid it being a bottleneck.

C. *Wired vs Wireless*

We wanted to see how h2's performance would be impacted with a wired vs wireless connection. Looking at Figure 6, we can see that h2's performance is more or less constant over wired or wireless connection. Wireless shows no trend, likely due to high variation of wireless connectivity. In the future we would like to compare these results with wired and wireless data for h1.1 to see how much of a performance gain we should expect from using h1.1 vs h2 wirelessly.

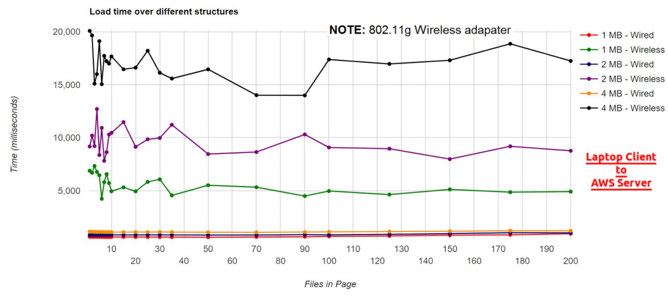
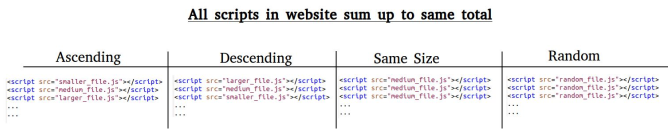


FIGURE 6. WIRED VS WIRELESS CONNECTION TRANSFER TIME

D. *Structure*

When we talk about structure, we talk about the order the files are downloaded from the server. Figure 7 shows what this structure looks like on our automatically generated sites. On Figure 8, we can see that transferring a file in descending order versus ascending order is much faster in beginning. A 2MB website shows 22% increase in performance, a 2MB site shows 13% increase and lastly, a 1MB site shows a 6% increase. This shows that there is a linear increase in performance, doubling size of website results in doubling the percent difference between descending and ascending structure. We believe this is due to slow start in congestion control, and the fact that h2 provides full utilization of the TCP pipeline when sending larger files. The difference in transfer times when sending between 20 to100 files was not found to be significant.



All scripts in website sum up to same total

Ascending	Descending	Same Size	Random
`<script src="smaller_file.js"></script>`	`<script src="larger_file.js"></script>`	`<script src="medium_file.js"></script>`	`<script src="random_file.js"></script>`
`<script src="medium_file.js"></script>`	`<script src="medium_file.js"></script>`	`<script src="medium_file.js"></script>`	`<script src="random_file.js"></script>`
`<script src="larger_file.js"></script>`	`<script src="smaller_file.js"></script>`	`<script src="medium_file.js"></script>`	`<script src="random_file.js"></script>`
...	...	...	...
...	...	...	...

FIGURE 7. EXAMPLE OF ASC,DESC,RAND, SAME SIZE REQUEST STRUCTURE

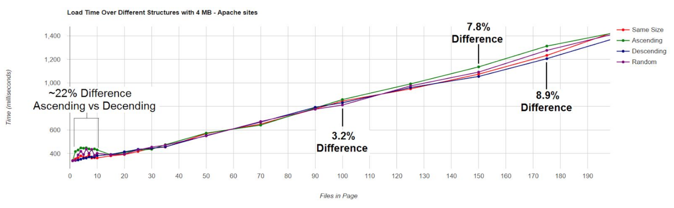


FIGURE 8. PERFORMANCE OF STRUCTURE ASC,DESC,RAND, SAME SIZE

E. *Server Push*

We observed no advantage between server push on and off when enabling it and disabling it for Apache. Nginx did not have server push so we compared its data with no server push against that of Apache with server push on and off. We found no advantage between using Nginx or Apache.

We believe this is due to the fact that our generated websites did not contain links on the header specifying which files to preload on the client through server push. We thought this was a server side optimization which would be done automatically. In the future we would like to explore adding this information into the packet header in order to more accurately test server push.

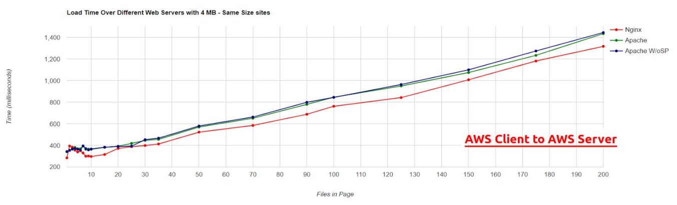


FIGURE 9. NGINX VS APACHE PERFORMANCE

VII. CONCLUSIONS

Although were not able to identify a clear optimum number of files, file sizes or structure that would provide the best performance for website, we were able to determine general principles for developers to follow. In general, we would recommend keeping your website or web app as small in size as possible, structuring it in descending order, and creating the least number (without concatenating) of files whenever possible.

Although this suggestion may seem to imply concatenating files, we would advise against it as h2 does not suffer from head-of-line blocking that makes concatenation in h1.1 advantageous. Without concatenating files, caching can improve the overall performance.

h2 is best suited for websites with a large amount of required files that are updated relatively frequently. This is due to h2's performance increase over h1.1 in regards to a large number of files, and caching.

h2 shows a clear advantage over h1.1, but its adoption will continue to be slow until developers determine that the performance of their website or web apps is being limited by h1.1, and the only solution is to adopt h2. As more developers begin to utilize h2, clearer best practices and standards will be defined.

Another contributor to its adoption will be user satisfaction due to load times between competing applications. We hope that developers will take the results we have generated to begin using h2 to its full potential in order to create a faster internet for the future.

VIII. Discussion

A. Limited H2 Customizability

When we started this project, we set out to determine the inflection point at which h2's performance overtakes that of h1.1 for certain variables. Our research aimed to benefit developers working on new websites and web apps. By determining the optimum number of files, file sizes, file size transfer order, and server environments for h2, we hoped to create best practice guidelines for these developers.

We researched ways to edit configuration of h2 but quickly learned that there weren't many settings we could modify, there was only the option to enable or disable server push.

B. Raspberry Pi Limitations

From of our findings using raspberry pi, we discourage the use of budget hardware like the raspberry pi in future tests and research as its hardware limitations degrades the accuracy of the data. We believe the bottleneck to be due to the raspberry pi's inability to process the large number of packets on its 1Ghz quad-core processor, previously shown in Figure 5.

C. Scope and time constraints

Due to our initially ambitious project goals and time constraints of the semester, we were not able to test the h2 protocol using firefox. This was also made more difficult by the fact that firefox does not have a headless browser in order to run our tests.

A large part of h2 is flow control which is implemented differently by each browser. For example, a browser can choose to request html and css files first so that the page can load before requesting JS libraries and images. Because of the variability of in the file priority implementation we opted to solely using JS files for out data.

3G could not be tested due to: time constraints, automation development cost, and cellular data charges.

IX. Responsibilities

We tried to distribute the work evenly as a team. Christian worked on the BASH website generator script. Spencer worked on setting up the raspberry pi's, laptop and amazon web services with chrome headless har capturing script, nginx, apache and MySQL DB, as well as the outlier detection script. Emmanuel and Spencer worked on the har parser and upload script, the database design, as well as scripts to generate the google charts to visualize the data.

Acknowledgements

Thanks to my dog Cosmo for keeping me company. Thank you to coffee.

References

[1] HAR 1.2 Spec | Software is hard. (2017). Softwareishard.com. Retrieved 14 March 2017, from http://www.softwareishard.com/blog/har-12-spec/

[2] SPDY. (2017). En.wikipedia.org. Retrieved 14 March 2017, from https://en.wikipedia.org/wiki/SPDY

[3] XVFB. (2017). X.org. Retrieved 2 April 2017, from https://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml

[4] Making the Web Faster with HTTP 2.0 - ACM Queue. (2017). Queue.acm.org. Retrieved 23 April 2017, from http://queue.acm.org/detail.cfm?id=2555617

[5] HTTP/2 Will Change the Internet... Eventually. Lonn, R., & Lonn, R. (2016). Bluemix Blog. Retrieved 24 April 2017, from: https://www.ibm.com/blogs/bluemix/2016/12/http2-will-make-the-internet-faster/

[6] HTTP/2 vs. HTTP/1.1: A Performance Analysis. (2017). YouTube. Retrieved 24 April 2017, from https://www.youtube.com/watch?v=0L5Q_897fwk