CSE 534: FCN - Project Report

HTTP/2 Prioritization using H2O server

Adarsh Alangar (112026947)
Dhruva Gaidhani (111971181)

1. Introduction

A significant amount of research is being carried out in Page Load Time optimization which is an important web performance metric. In this project, we study in detail how prioritization of resources by HTTP/2 affects web performance. We replicated the experiments made by Wijnants et.al. [1] to corroborate the results published and to obtain a deeper understanding of resource prioritization.

The new HTTP/2 specification includes dedicated resource prioritization provisions over single, well-filled TCP connections. HTTP/2 aims to solve Head Of Line blocking by coalescing resources on a single underlying TCP connection with an explicit resource prioritization mechanism based on dependency relationships and weights. This implies that the client gets to make the request with priorities defined by the client as per the need. These priorities are suggestions and can be ignored by a peer [7].

In this project we elucidate and augment the results presented by Wijnants et.al. [1]. We find that the results for Google Chrome are accurate, whereas Mozilla Firefox's dependency tree is ever so slightly more complex than presented. Largely, our finding are in line with the authors, although there are subtle differences. We also present a visualization tool to generate a dependency tree from server logs.

2. Problem Statement

We set three primary objectives for the project:

1. Replicate the environment created by the authors with the H2O server and list down the steps to do so. This would allow any future work to leverage the effort made here.
   Appendix I
2. Understand and elucidate HTTP2 priorities and dependency trees. We use Google Chrome and Mozilla Firefox as they are browsers with significantly orthogonal approaches [1].
   Section 5
3. Re-create and corroborate the results presented, comparing various dependency trees and the load times that they lead to.
   Section 6

3. Approach

We perform most of our analysis and modifications on the server side. We've analysed and modified the configuration of browsers in two instances:

1. Setting up the browsers to log PLT with no cache, cookies or history (section 6).
2. To analyse Mozilla Firefox dependency trees (section 5).

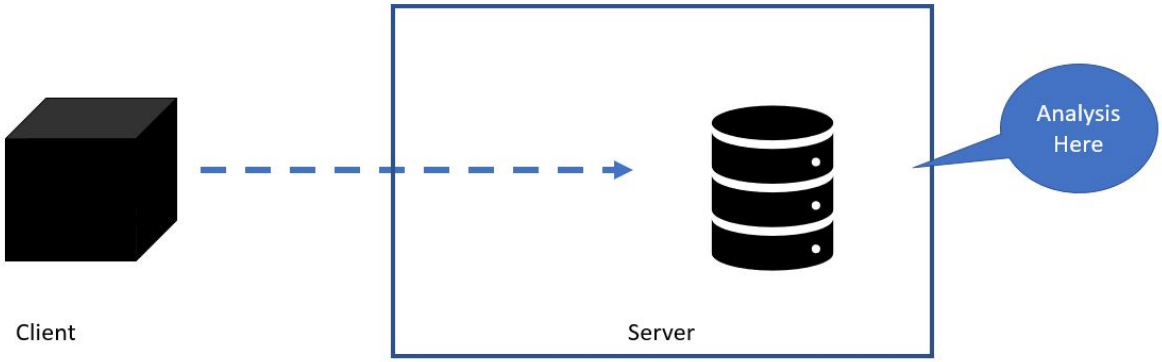


Fig 3.1

Fig 3.1 gives an overview of the approach. To understand the working of browsers instead of looking at their source code, we only look at message coming out from them. This allows us to focus on the priority and stream packets, ignoring other browser intricacies. In most cases the browsers are just black-boxes for the sake of simplicity and experimentation.

4. Set up

To set out, we choose the path with the least resistance. We use similar Operating Systems (Debian "jessie"), same code base and similar machines to the ones used in the original paper. We have a client server model running on the same virtual machine. We first initialize the server to a plain H2O v2.1.0 instance which has absolutely no modifications and hit it with the client, all the while analysing the behaviour of the communications that take place and logging the results along with metric *loadEventEnd*. We perform a similar hit with a modified H2O as mentioned in the paper and carry out the same analysis and compare the results. We draw out the differences between the dependency graphs that are built in both of these cases.

Appendix I lists of the exact steps taken (including commands used) that are required to get H2O up and running. This fulfills objective 1.

Another important aspect of this project is to identify how this server-side implementation ignores the client-issued prioritization directives, in accordance with the HTTP/2 specifications and instead drafts a custom dependency tree according to a new scheduling logic.

Visualization script: We also built a tool to help visualize the dependency between streams. This uses some logging statements introduced by us in H2O as the input. The output of the tool is used in section 6.

5. HTTP2 frames, streams, dependency trees and browsers!

- The basic protocol unit in HTTP/2 is a *frame*. Each frame type serves a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses; other frame types like SETTINGS, WINDOW_UPDATE, and PUSH_PROMISE are used in support of other HTTP/2 features [7].
- A *stream* is a bidirectional flow of frames within the HTTP/2 connection [7]. A stream is a logical segregation of a connection. Every stream is identified by a unique stream ID which is unique in the connection.
- HTTP/2 optionally allows browsers to specify a dependency among streams therefore constructing a dependency tree. This dependency tree is dynamic and follows an important rule that a child stream can

be used to send data only if the parent is closed, idle or blocked. Additionally, a *priority* between 0 and 255 can be assigned to every link between a parent and child. This priority is a hint or a suggestion to the server asking it to multiplex resources based on the priority.

5.2 Stream Identifiers

A stream is uniquely identified in a connection using it's ID. An ID cannot be repeated. A stream initiated by the client should have an odd ID, and server initiated stream should have an even ID.

More importantly, stream should always be created in an increasing order of their ID. The ID of any new stream should always be greater than any existing stream ID.

Stream 0 is the default stream and is always the root in the dependency tree.

5.2 Transferring a tree from a client to a server

In the internet domain, browser's build and transfer dependency tree to the server. This does not always have to be the case as servers are allowed to decide priorities too.

A peer has three HTTP/2 frames that allow it to create or re-prioritize a stream:

- •HEADER frame: Always open a new stream, additionally send traditional HTTP headers
- •PRIORITY frame: Create or re-prioritize streams
- •PUSH_PROMISE: Used during server push by the server. (cannot send priority and is assigned a default dependency to the stream in which it is sent)

We primarily look at the HEADER and PRIORITY frame. They contain a *stream dependency* and a *weight (*priority*)* field. A client computes the dependency tree and sends either a HEADER or a PRIORITY frame to the server, therefore transfering the tree one node(stream) at a time. This means that the server builds a tree based on each packet received. It is at this point that H2O chooses to ignore the priority and dependency received. It then overrides the two fields with desired values.

5.3 Google Chrome - Default tree

Google Chrome implements a dynamic First Come First Serve (FCFS) algorithm. We were able to observe and verify the claims made in the paper. Below (Fig 5.3.1) is a comparison of the trees given in the paper against a tree produced by the visualization tool built by us. We observed buckets similar to the ones mentioned in the paper.

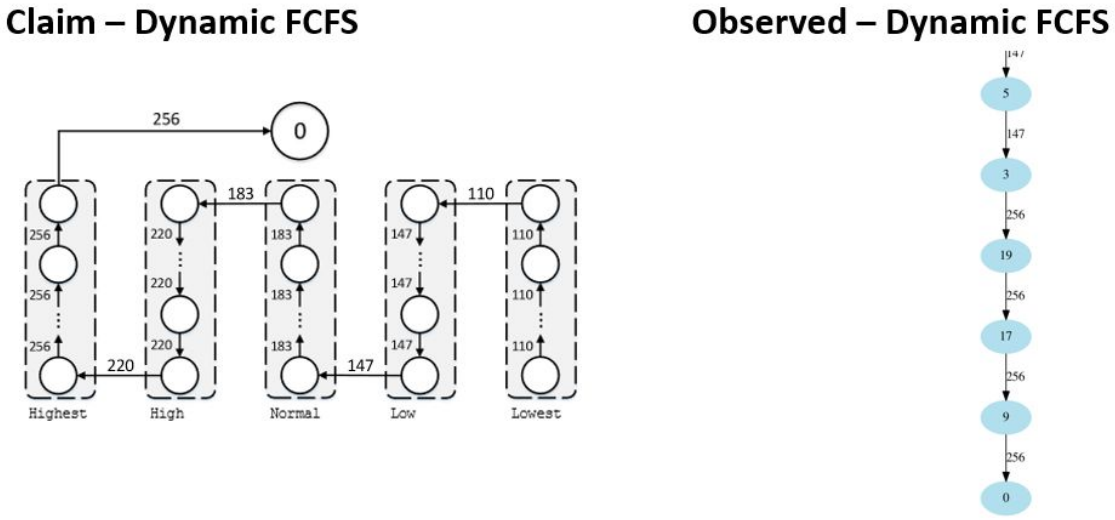


Fig 5.3.1

Below (Fig 5.3.2), we see an instance of the dynamic nature of Chrome. Let's assume that at some execution step there are two existing with priorities 147 (usually images) depending on the root. At this point a request for a node with priority 220 (JS before the first image) is received. Chrome would immediately push this to make the new node a child of the root. This dynamicity allows Chrome to ensure that higher priority noders or objects are prioritized more.

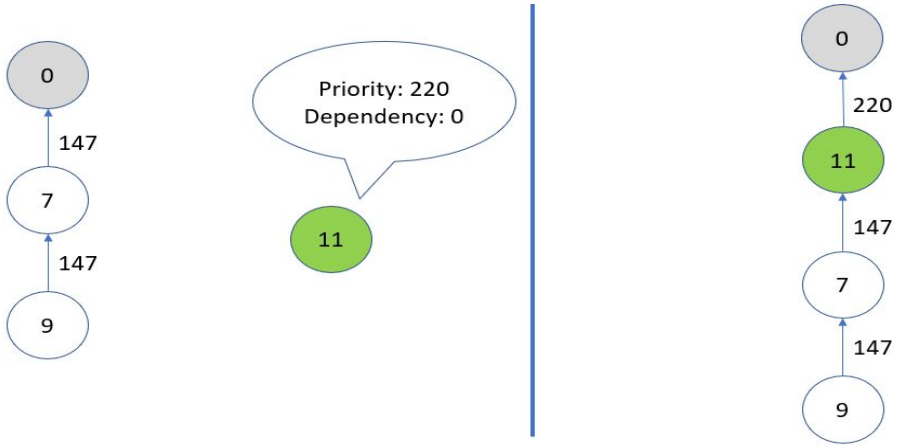
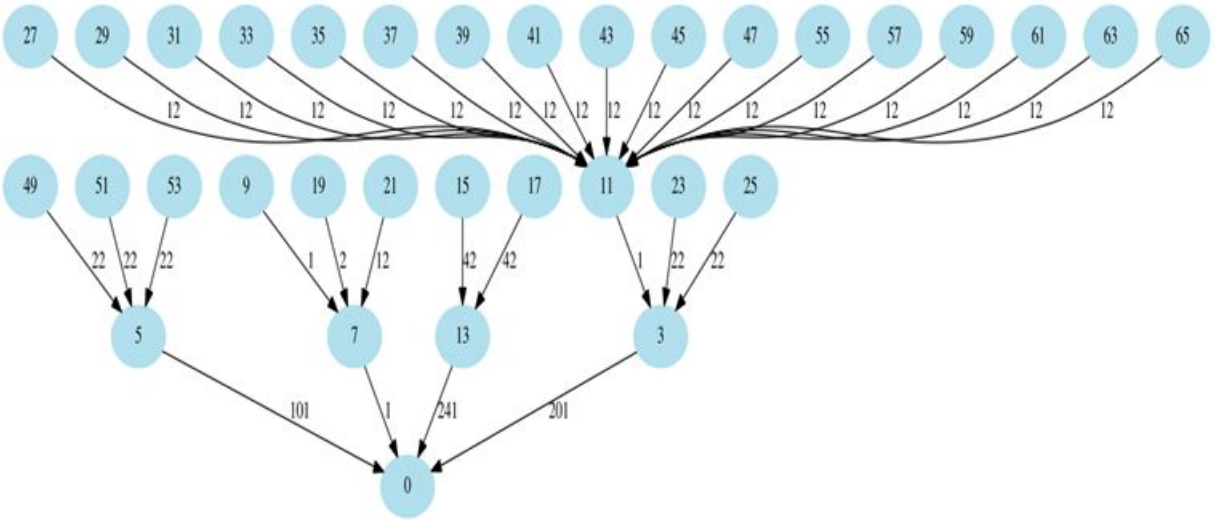


Fig 5.3.2

5.4 Mozilla Firefox - Default tree

Firefox's take on building a dependency tree is quite different from Chrome. While Chrome uses a dynamic FCFS algorithm, Firefox builds a complex tree structure. Before the transmission of data, Firefox creates 6 phantom nodes. Phantom nodes are not used to transmit data, but serve as idle streams on which other stream depend on. They can be thought of as the "foundation of the tree". Here our observation differs from the authors. We observed 6 phantom nodes compared to 5 reported in the paper. A source code analysis of Firefox confirms our findings [8]. Below (Fig 5.4.1, bottom) is the correct version of Firefox's dependency tree layout (additional stream is numbered 13).



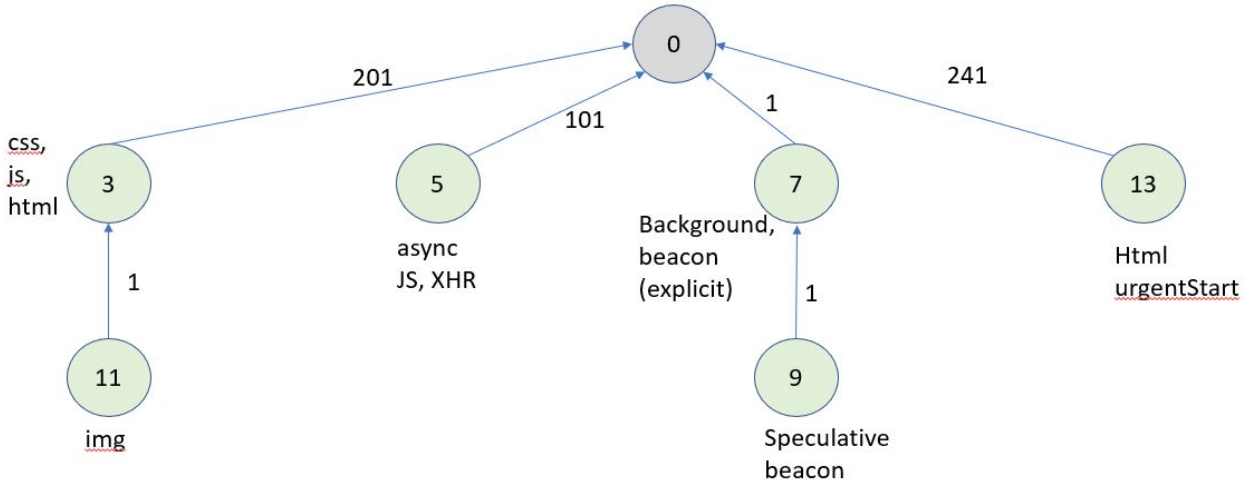


Fig 6.4.1. Top: From the custom visualization tool
Bottom: Only the root and phantom nodes

5.5 Custom Dependency Trees

The author's present 4 custom and two naive dependency trees. The customs trees as follows:

- Serial for Firefox
- Serial+ for Firefox
- Parallel for Chrome
- Parallel+ for Chrome

These four are well described in the paper and are implemented faithfully. They have been omitted here.
We validated Serial for Firefox and Parallel for Chrome as a part of this project.

5.6 Naive Dependency Trees

Two extremely simple approaches, First Come First Server (not dynamic) and Round Robin (no weights/priorities) are also evaluated. These are straightforward and therefore not discussed in detail. The analysis for this has been done in the following sections.

6. Survey Evaluation Setup

In order to verify the results obtained by the authors in their original survey, we set up a Linux Debian Jessie (8.11) virtual machines with 2GB of dedicated RAM and 1 dedicated CPU core. This virtual machine hosts both the server (H2O) and the client which in our case were Firefox (version 54) and Chrome (version 58). We set Page Load Time (PLT) to be the base metric for our tests. To evaluate PLT, we use the inbuilt tool in the browsers consoles that allows us to perform network analysis. Having a combined client-server model greatly reduces any network related uncertainty. To account for this and make the results more realistic we added various network delays ranging from 0 to 1000ms. Another reason behind doing so is that it actually allows the server to use the specified scheduling algorithm, which without a delay, services all the requests so fast that the scheduling algorithms do not come into play.

Unlike the base paper, we lack access to the Speeder framework [4] since it is closed source. But based on the results obtained in the paper, we made configuration changes to the browser. The configuration files for both the browsers were changed to remove caching of data, storing history and cookies. This allows a fresh start every time a request is fired by the browser giving an independent set of results. We consider the LoadEventTime as the page load time which is specified by the browser in the console. From the test corpus of the authors, we

limited ourselves to 20 websites. We picked at least 6 websites from each category that the authors used. Out subset of web pages included:

- 6 Light Weight Websites
- 6 Medium Weight Websites
- 8 Heavy Weight Websites

The websites are queried five times with a cleaned history and all the records are maintained and tabulated. An excel file documenting the results for a 1000ms induced pseudo-delay run is submitted alongside this report. Below is an analysis of the test corpus. These set of pages are a subset of the original database of pages which have been selected from Alexa Top 50 and Moz Top 500 rankings. This subset is so chosen because it provides just as good a sampling of the original web page set to get a good distributed of the page weights and resource counts. The following graphs provide a brief overview of the number of resources and weights of the websites under consideration. The graphs signify the spread distribution of the test Corpus.

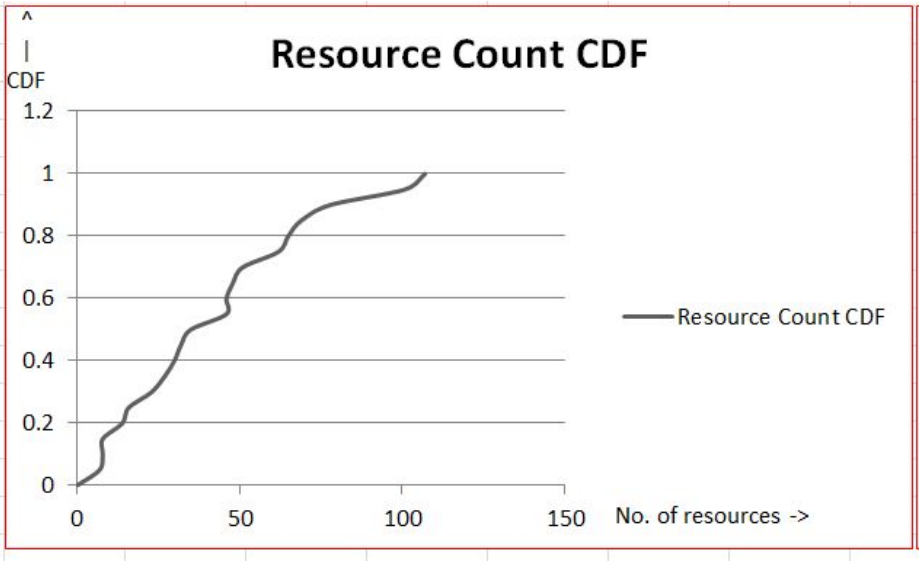


Figure 7.1: CDF of asset count

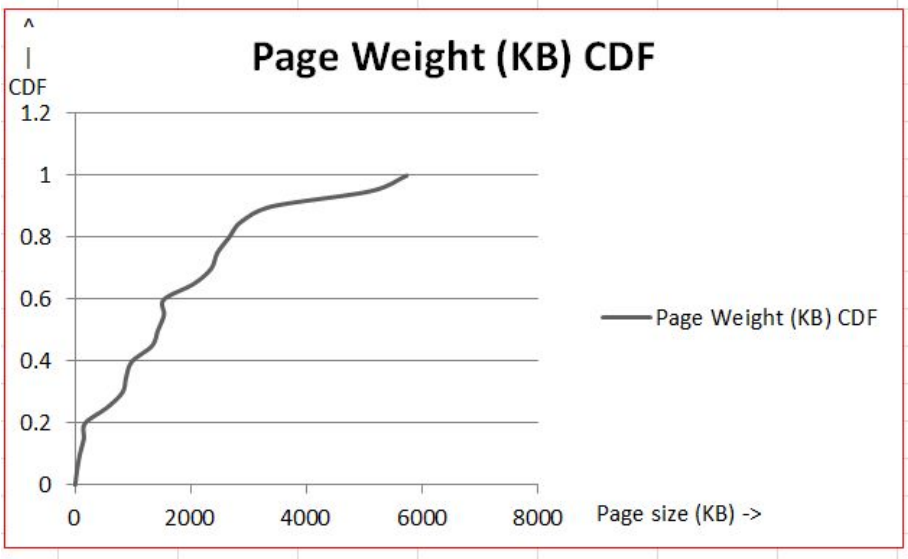


Figure 7.2: CDF of Page weight

7. Survey results

- We recreated the virtual working environment used by the authors - including the exact same operating system version, memory and CPU cores (to the best of the host system's abilities).
- With the modified H2O server, compiled from the source, up and running on a virtual Machine as with specifications as stated above we performed our extensive survey of the Page load times for the list of websites mentioned in the Excel file which gave us the following results:

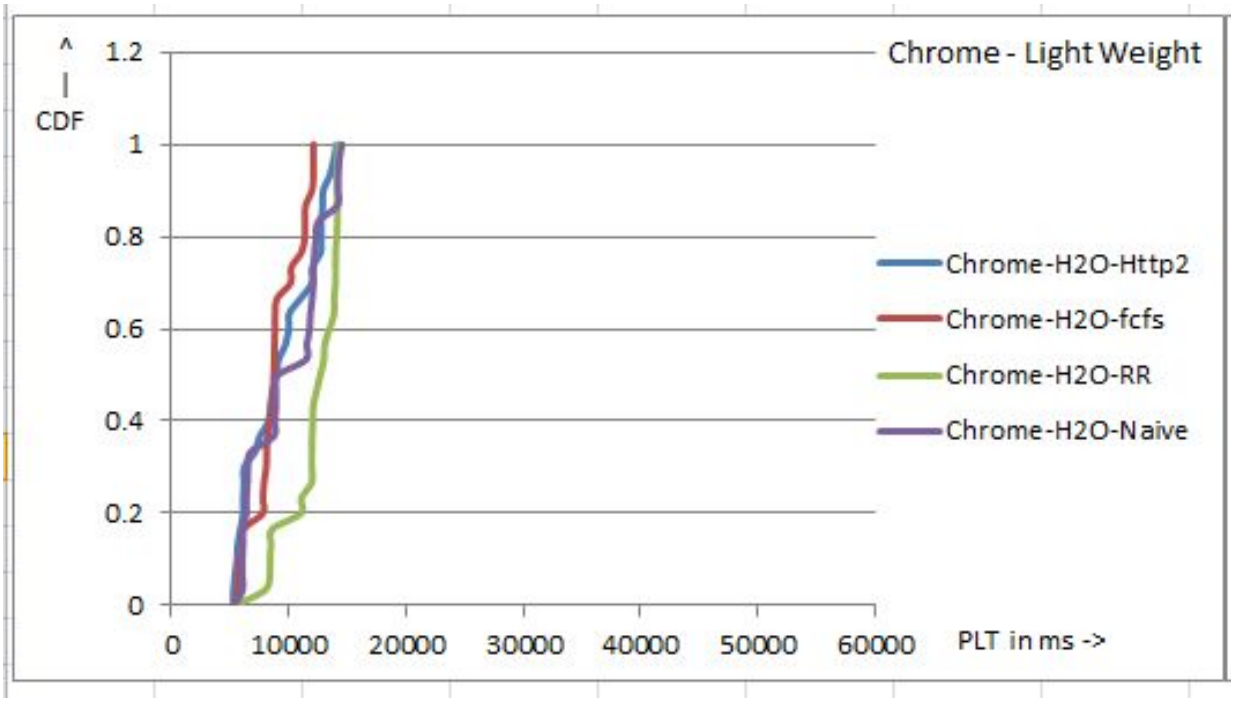


Figure 8.1: CDF of Chrome against light weight pages

The HTTP/2 specification includes dedicated resource prioritization provisions, to be used in tandem with resource multiplexing over a single, well-filled TCP connection. This section of the experiment was mainly aimed at understanding how do these two most popular browsers apply this and how is the page load performance affected for both.

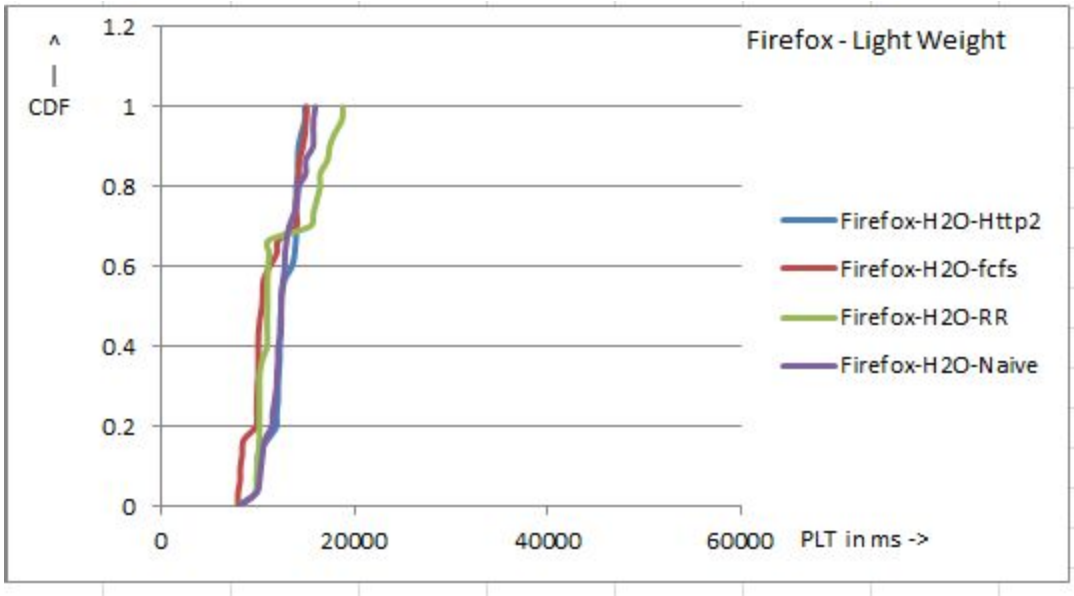
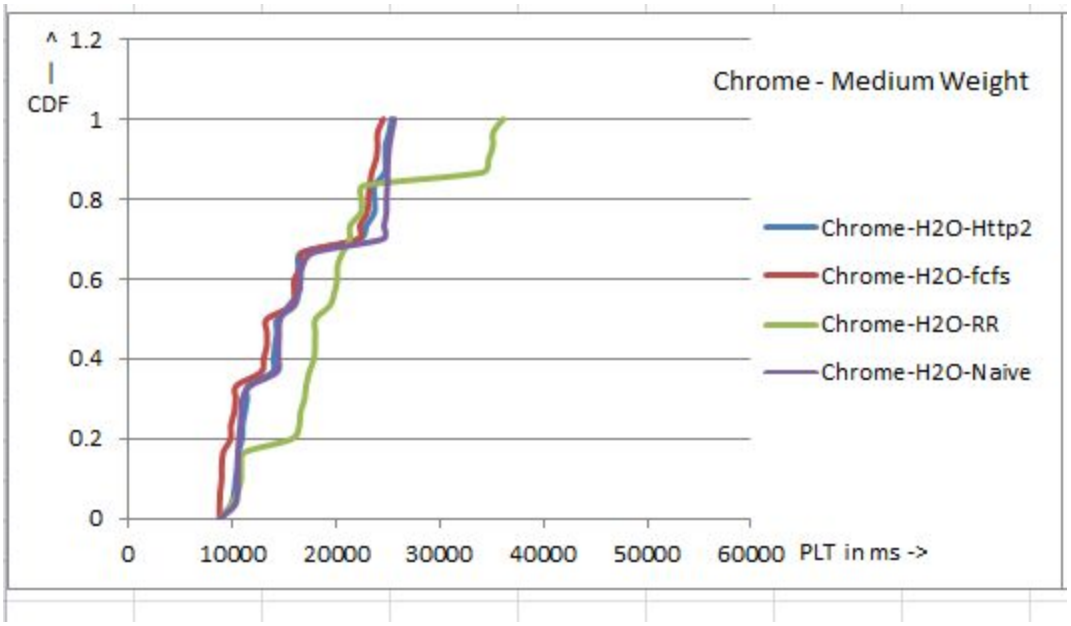


Figure 8.2: CDF of Firefox against light weight pages

Light weight page load results:

- For chrome, just like the paper we observed that the naive Round Robin algorithm performed the absolute worst. We hypothesize that this can be attributed to the fact that the websites under that range had larger than average size per object. Round robin seems to result in contention among higher priority objects with lower priority ones, causing an increase in the PLT. The paper mentions that the authors were surprised that HTTP/2 specification promotes Round-robin algorithm. We went through the specification under question but found no mention of round-robin specifically.
- We see that FCFS performs comparably to default priority scheduling algorithms of the browser, however as we have seen that Chrome itself is based on a variant of a dynamic FCFS priority scheduling algorithm, this scheduling yields slightly better results as we progress further.
- As opposed to this, for round robin, firefox does not see a huge difference in performance right off the bat. But with progressing page size, round robin performs badly for websites with a large number of resources. The FCFS scheduling algorithm works better for smaller websites and with time.



Medium weight page load results:

- As before we see that Chrome's performance significantly falls with round robin. For medium weight pages we see a huge variance in the page load times. The paper hypothesizes that this is because of a number of factors which includes but is not limited to the actual source code of the website.

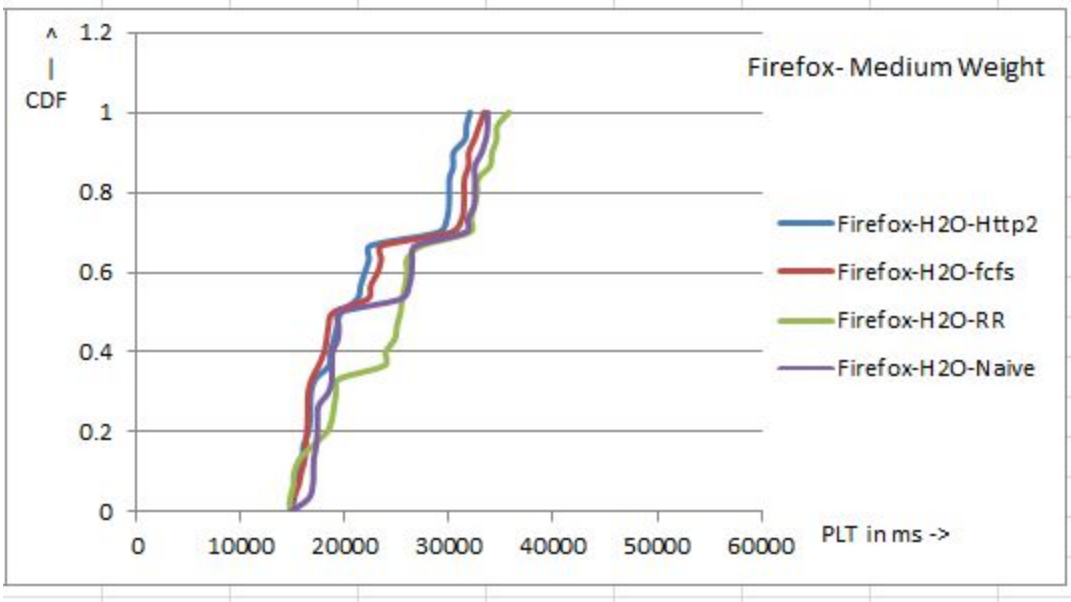


Figure 8.4: CDF of Firefox against medium weight pages

- Naive FCFS performs the best but there is not much difference when checked against the default algorithm.
- For Firefox, we see a steady change rate for all the scheduling algorithms. While Round robin still performs the worst amongst the four, we see that the default scheduling algorithm takes a lead over FCFS from before.

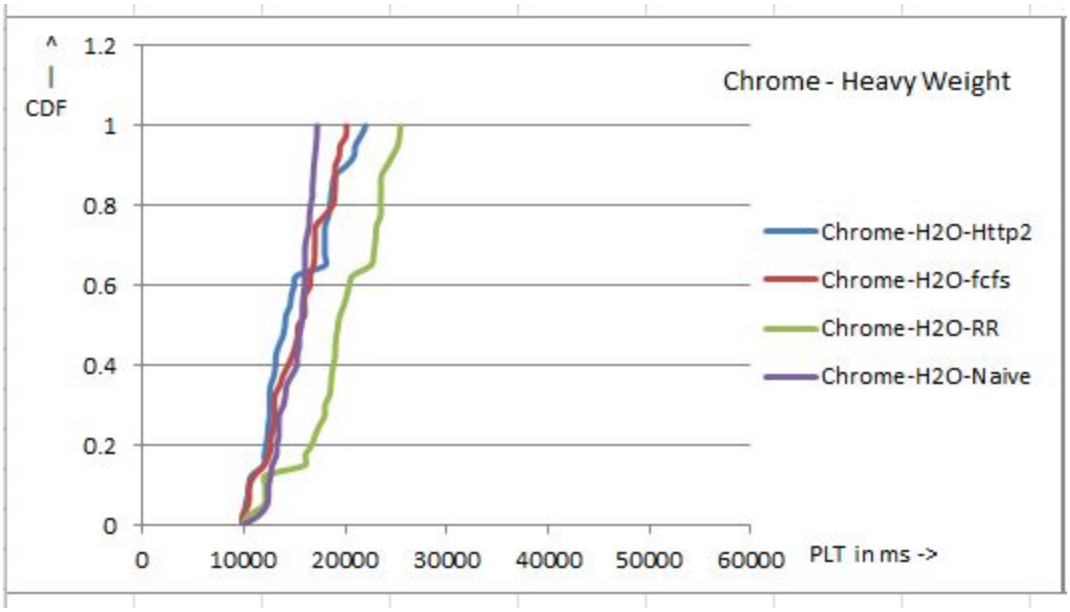


Figure 8.5: CDF of Chrome against Heavy weight pages

Heavy weight page load results:

- We find that prioritization algorithms introduce a significant change here as the graph is seen to branch out at the ends.

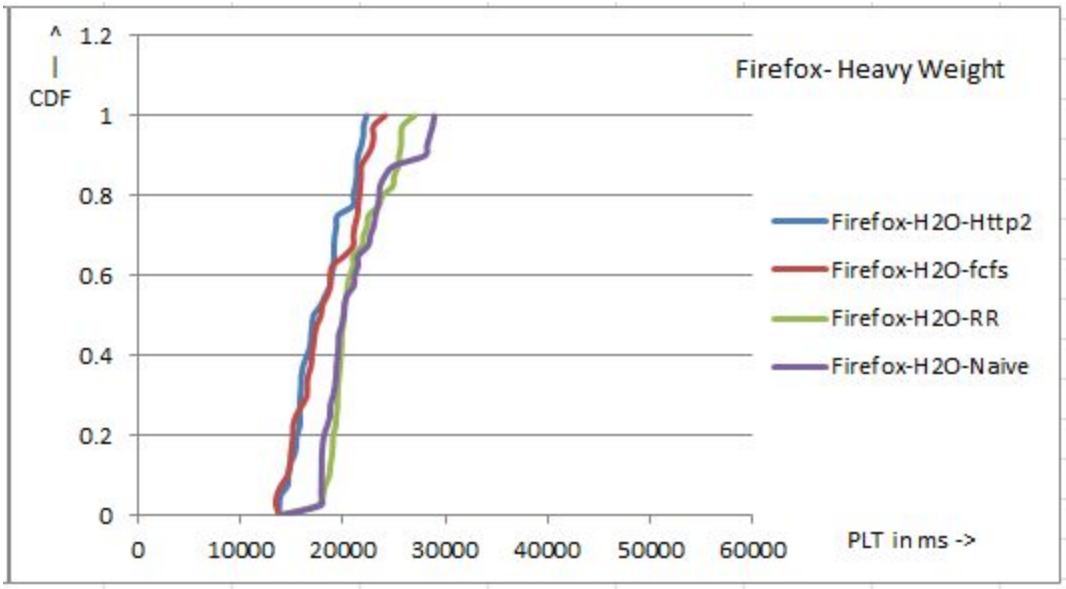


Figure 8.6: CDF of Firefox against Heavy weight pages

- For increasing weights, for both Firefox and Chrome the default scheduling algorithm performs well, but with increasing weight of the pages, FCFS falters slightly.
- For Firefox, with increasing average size per object, the default scheduling algorithm performs the worst which can be attributed to the complexities of building an elaborate dependency trees.

8. Conclusion

In this project we were able to elucidate the ideas behind the working of HTTP2 resource prioritization and its dependency trees. In line with this, we verified the dependency trees of Google Chrome and Mozilla Firefox (albeit with minor changes). Our tool helps visualize complex dependencies that browsers build by reading their logs. Additionally, we list down the changes needed to setup a H2O server for future works.

A subset of the test corpus was put through the exact same Page Load Time tests as the paper. The PLT for our test se was analyzed and compared with the authors' findings along with the hypothesized claims regarding the reasons behind why a certain page probably has the observed load time. Our findings were very similar to that of the authors, which have been explained in detail in the prior section. We also find that the speculations of the authors, regarding the resources or weight of a website and how it likely contributes to the PLT have to be true given the correlation between our graphs and theirs.

Some extensions to this project include verifying the results for cellular networks and for cable networks with an induced packet loss. To run tests against the more advanced algorithms and create a data set for serialized and parallel plus algorithms to verify the claim that complex scheduling algorithms actually result in over 15% Speed up in the Page Load process.

It will also be interesting to identify why the FCFS scheduler performed so well. More research needs to put into this to verify the aforementioned claim.

8.1 Justification for additional 30%

- Creating an independent visualization tool to understand client side functioning without browser modifications.
- Analyzed weight of the page, the number of objects, the type of the objects and the order in the website code to identify its correlation with our Page Load Times and verify if they were in line with the Authors' thinking.

- Identified an additional “branch” in the Firefox dependency tree and verifying this find through source code analysis.

9. Source Code, Data and Results

https://github.com/adarshdec23/h2o_custom_priority

10. References

[1] https://speeder.edm.uhasselt.be/www18/files/h2priorities_mwijnants_www2018.pdf
[2] https://speeder.edm.uhasselt.be/www18/
[3] https://www.webpagetest.org/
[4] https://speeder.edm.uhasselt.be/
[5] https://developers.google.com/web/fundamentals/performance/http2/
[6] https://evertpot.com/http-2-finalized/
[7] https://http2.github.io/http2-spec
[8] https://dxr.mozilla.org/mozilla-central/source/netwerk/protocol/http/Http2Stream.cpp#1320

Appendix I

We started from a fresh installation of Debian 8.11. This is the exact version used by the authors.
Download the modified source code from:
https://speeder.edm.uhasselt.be/www18/files/H2O_H2priorities_Custom.zip

Before we can proceed with the set-up of H2O, we need to ensure that we all the right tools in place. So we install the C/C++ toolchain required to run H2O using command

```
sudo apt-get install build-essential
sudo apt-get install autoconf
apt-get install libtool
```

We now install dependencies required by H2O. To do this, run the following command

```
apt-get install locate git cmake build-essential checkinstall autoconf pkg-config
libtool python-sphinx libcunit1-dev nettle-dev libyaml-dev libuv1 -y
```

H2O also requires Perl to run. The following commands takes about an hour to run, so run it when there’s time

```
sudo perl -MCPAN -e 'upgrade'
```

One more dependency that H2O has is on wslay. Download git clone
https://github.com/tatsuhiro-t/wslay.git

The docs in the homepage of wslay should be followed. If this throws an error, that’s okay. Undo the changes in the following commit and it should be fixed
https://github.com/tatsuhiro-t/wslay/pull/49/files

A couple of more dependencies and we should be all set!

```
sudo apt-get install libssl-dev
sudo apt-get install zlib1g-dev
```

At this point we have all the tools required. Time to ‘make’ H2O. Goto the downloaded location and unzip it. Then run cmake:

```
cmake -DWITH_BUNDLED_SSL=on .
```

This should run fine. After this we ‘make’ and install it. These two steps need to be repeated after every modification made to the source code.

```
make
sudo make install
```

Now we can run the server! Sample command to run it:

```
sudo h2o -s "rr" -w -c h2o.conf
```

The --help option is very useful and the “wordy -w” command gives a detailed description of the log files!