Connection Booster

Meryem Essaidi, Joomy Korkut, Anastasiya Kravchuk-Kirilyuk

Abstract

As the user base around the world grows larger, improving browser performance is more critical than ever. We propose Connection Booster - an app that uses a formula based on our empirical observations to optimize the number of parallel TCP connections which the browser opens to acquire the resources on a page. In the following work, we describe the advantages and mostly disadvantages of the Chrome API in the implementation of our app, and explain the design decisions we made based on our findings about the Chrome API. We also analyze the empirical gains in terms of performance when using our app as opposed to the Google Chrome browser, and provide ideas for further improvements.

1 Introduction

Most browsers already employ techniques to boost performance by optimizing TCP connections. For example, Google Chrome uses TCP-PreConnect in order to reduce latency delays and slow-start penalties [3]. Another way to improve browser performance is to make use of TCP parallel connections. By opening more connections, the browser can send and receive more packets via the different streams, which can lead to an increase in bandwidth and also alleviate congestion [5]. When the workload is divided across different streams, there is a smaller chance that an overloaded stream will trigger TCP congestion control. However, if too many parallel connections are open and are not utilized to their full potential, resources are wasted that could be better used elsewhere. As a solution to this problem, we propose a Google Chrome application that automatically calculates the optimal number of TCP parallel connections to run to maximize performance. We define performance as the time it takes (in ms) to load the page.

2 Design

Currently, most browsers already limit the maximum number of parallel connections to the same server allowed at one time. For example, Google Chrome's limit is 6 [11]. However, the Google Chrome API allows opening and closing of TCP and UDP connections. It also allows plugins that have the necessary permissions to inspect the resources in a page. Before starting the project, we conjectured that this meant that a plugin can open parallel connections to load resources.

Although some optimization techniques are already built into Google Chrome, our project aimed to enhance performance even further by parallelizing resource loading in a better way. At the very least we hoped to provide a playground in which we could test what parameters should be taken into account in deciding the optimal number of parallel connections.

Our initial plan was to make our project a Chrome extension, however Chrome does not allow the usage of the TCP sockets API for extensions, it only allows it for "Chrome apps", which are applications that run on the Chrome infrastructure. In other words, Chrome apps are basically web pages that run in a separate window, outside of the browser. The realization that we had to write a Chrome app if we wanted to use TCP sockets limited the project quite a bit from the very beginning, since it meant that we had to implement a mini browser application as a web page.

The example web page we have built consists of two main parts, an address bar and an embedded page which will show the page we want to load. The address bar simply consists of a text input field and a button that loads the page at the given address. For the embedded page, however, we had to choose between a `<webview>` [2], which is a special element provided by the Chrome API that is used for embedding pages in Chrome apps, and a classic HTML inline frame, also known as an "iframe".

We chose <webview> over iframes because <webview>s run in a process separate from our app, and it does not give the page the same permissions as our page. This is a security benefit, since our app requires extensive permissions. However, <webview>s are a bit more restrictive than iframes, in that our app is only allowed to manipulate the DOM structure of the embedded page through very limited injection rules. These rules only allow modification of the DOM structure by passing a piece of JavaScript code (either as a string or a separate file). This means we cannot use the Chrome API inside the <webview>, which is useful for security purposes, but restricting for experiments like ours. The only way to circumvent this is to send messages between our app and the <webview>, but that requires injecting message sending code into the embedded page [1].

We identified two methods to put content in a <webview>:

1. Setting the src attribute of the <webview> element to the web page URL the user types in the address bar
2. Setting the src attribute of the <webview> element to a data URI [7]

We initially wanted to go with the first approach, but it turned out that there exists no JavaScript event that can fire right after the DOM is loaded, but right before the resources start to load. The original idea was that our app would use raw TCP sockets to load the resources and then it would inject the resources into the <webview>. However, this was impossible due to the limitations of JavaScript load events. In our final attempt to make the first approach work we explored the events available through Chrome's WebRequest API, however this API was not available to use in apps like ours. Ironically, we would have been able to use this API if we were implementing an extension.

This meant we had to follow the second approach, which was a much bigger task than we initially envisioned. First of all, we implemented a small portion of the HTTP protocol. Our app opens a TCP socket to the host at port 80, connects, sends an HTTP request, and receives a HTTP response in many chunks. The first chunk always contains the HTTP response headers. Our app parses these headers and further collects the data received in each chunk. Then our app injects that data manually into the <webview> using the data URIs.

It would be helpful to revisit data URIs before we describe the challenges of our approach. A (base 64) data URI is when a URL has the format data:<mediatype>;base64,<data>. The media type is the MIME type of the file, and data is the actual content of the file in base64-encoding. For example, if we have a file the content of which is just d4 { background: pink; }, we can encode the entire file as a URL using data URIs, as data:text/css;base64,I2Q0I..., where the dots represent the rest of the file in base64-encoding.

The advantage of setting the src of the <webview> to the the URL in the address bar is that it could be used as the "base URI" of the document, which would resolve all the relative URLs in the file automatically. However, in our approach we must have a data URI for the <webview> src attribute, which means the base URI would not be correct and the relative URLs would not be resolved correctly. There is a field called baseURI in the document object, however, it is not assignable. Therefore, our app has to construct and traverse the DOM structure of the page it is going to show, and change all the URLs to their fully qualified versions. In other words, all relative URLs like ./image1.jpg must become absolute URLs like http://domain.com/image1.jpg.

At this point in execution, our app has loaded the webpage into the <webview>, rewrote the URLs from relative to absolute, but it has not yet loaded the resources used within the page, such as the images, CSS style files, and scripts. Since our app has access to the DOM structure of the page, it can traverse the structure and get a list of all resource URLs it has to download. Once that is done, our app can reuse the HTTP protocol we implemented before.

In section 3, we explain how and why we automatically pick the ideal number of parallel connections based on our experiments, therefore we will not get into that in the design section. Once our app calculates this number, we can create that many parallel TCP connections to download all the resources in the page.

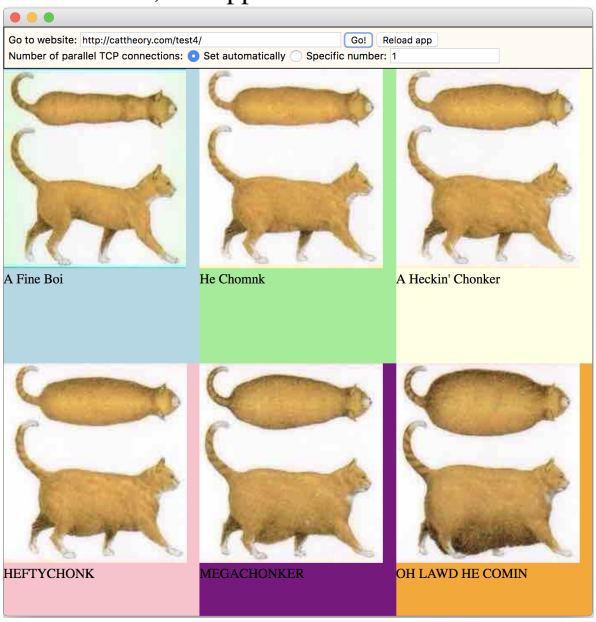
The next problem we had to solve was to put the loaded resources in the <webview>'s cache. Unfortunately, the <webview> API does not provide a way to inject content into a <webview>'s cache. It only provides a way to clear the cache [2]. This forced us to use data URIs again, but this time for populating the page with the resources. For example, if our

app finds an image element on the page, it will look at the `src` attribute, find the downloaded content of that URL, and substitute the URL with a data URI. This is far from ideal, since if a page uses a resource more than once, the content of the resource will be repeated that many times in the page content sent to the `<webview>`. Not to mention that data URIs have length limitations [7], which prevents our app from loading large resources, such as the large images in subsection 3.1.

Another consequence of this mandatory choice was that none of the resources get loaded to the page until all resources are downloaded. When the app finishes downloading a resource, it saves the file into an object that we call the "source map", which maps URLs to content. When all resources are downloaded, our app traverses the DOM structure and substitutes resource URLs with their contents, presented as data URIs. Then the app serializes the DOM, converts it into a data URI, and assigns that value to the `src` attribute of the `<webview>`.

A possible way to work around this hack could be to use the `<webview>` injection mechanism to update the DOM, which would let our app update the page incrementally, as soon as certain resources get loaded. However, since the `<webview>` content is determined by a data URI in the `<webview>`'s `src` attribute, our app needs a special permission to inject into it. Unfortunately, a Chromium bug prevents getting this permission for data URIs [4], which is why we could not use this technique.

At the end, our app looks like this:



2.1 Limitations

As a result of all these design choices, some of which we were forced into because of the shortcomings of the Chrome API and JavaScript events, we had to implement more than we thought we would. Therefore, we could only have a limited amount of functionality in our app. It can only handle the HTTP protocol, and all the resources also have to be in the HTTP protocol.[1] There is almost no error handling, all resources have to load without any error for the page to load at the end. Our app does not handle most of the HTTP protocol, such as `301 Moved Permanently` redirects, it can only work with the `200 OK` response. Clicks on a loaded page are not loaded in parallel the way our app does, only the "Go" button next to the address bar does the parallelization.[2]

Most importantly, our app only looks for images, stylesheets and scripts as the resources to download. If a resource is downloaded dynamically, or inside a CSS stylesheet, or a different kind of resource exists, our app cannot download it in parallel. This is because the Chrome API does not allow intercepting all outgoing connections from a page. Ideally our app should like to intercept and stop all outgoing connections and handle them in parallel. This is a serious limitation, but we managed to work around these to create a proof of concept application.

3 Evaluation

We decided to measure our app's performance by using load time of the page (in ms) as the measurement. Our original idea was to use aggregate throughput instead, but after receiving a helpful suggestion from the TA's, we realized that our target pages were webpages with quickly loadable resources, as opposed to pages with resources which were continuously loading, like a video stream. Therefore, we settled on load time since it better suited the usage of our app we envisioned. We then created four separate webpages with different numbers and types of resources in order to test the performance of our app in a relatively controlled environment. The first

[1] Not HTTPS. It would be possible to do but we did not have enough time.

[2] This is because message sending between the app and the `<webview>` is complicated. Clicks in the `<webview>` would have to send a message to the app to change the page.

test page, http://cattheory.com/test/, contains 4 very large images (1.5 MB - 5 MB each). The second test page, http://cattheory.com/test2/, contains the same images in a moderate size (180 KB - 800 KB each). The third test page, http://cattheory.com/test3/, contains only four CSS style files (30 bytes each). The fourth test page, http://cattheory.com/test4/, contains six CSS style files (30 bytes each) and six small images (4 KB each). In order to test how the different number of TCP connections affected the time (in ms) it took to load the page, we ran a trial for each possible number of connections (opening as little as one TCP connection at a time, and up to as many connections as there were resources on the page). For each set number of connections, we ran four trials and averaged the resulting load times. We also ran four trials loading the same webpage in an incognito Chrome browser window and averaged the results. We used an incognito window to make sure the caching of images did not interfere with the load time. In the browser measurements, we used the "Finish" time as the benchmark, due to the similar nature of our load time measurement in the app. We then compared the app load time average (per number of connections) with the browser load time average. The results of our experiments are represented in the following tables and charts. Note that the load time of the Google Chrome browser is represented as a dashed line across each chart to provide a means of comparison; this does not mean that the Chrome browser produces the same load time for any number of parallel TCP connections.

3.1 Large Images

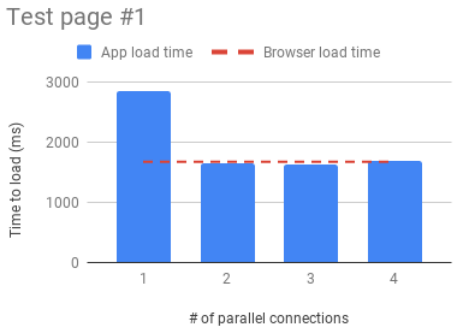


Figure 1: Large Images load times chart.

# of connections	Average load time (ms)
1	2847.5
2	1646.9
3	1629.3
4	1675.8
Chrome Browser	1672.5

Figure 2: Large Images precise load times.

From figures 1 and 2 we see that loading a webpage using only one TCP connection comes with a drastic increase in load time, which should be expected. When only one active TCP connection exists at any given time, the resources must be loaded in consequence one after another. It is clear from the data that in this case, 2 or 3 parallel connections give us the best load time result. In fact, even considering the extremely large size of the images on the page and the subpar design of the app (achieved through the vast incompetence of the Chrome API), our app still slightly outperforms the Chrome browser.

3.2 Moderate Images

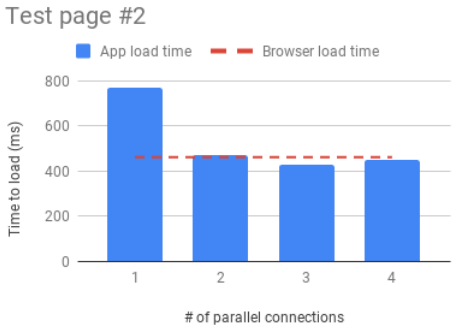


Figure 3: Moderate Images load times chart.

In figures 3 and 4 we see once again a familiar pattern: opening more than one TCP connection comes with benefits. In fact, it seems that opening even half as many parallel TCP connections as there are resources brings us extremely close to peak performance. Opening any more connections than that is not justified - the load time gains are minimal. Even though we might be tempted to open a separate TCP connection per resource and run all loads in parallel,

# of connections	Average load time (ms)
1	770.0
2	468.4
3	429.8
4	447.8
Chrome Browser	461.8

Figure 4: Moderate Images precise load times.

there is no need to do so. Furthermore, our app once again outperforms the Chrome browser.

3.3 CSS Style Files

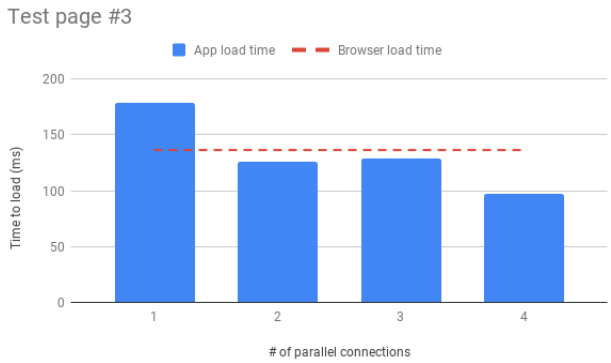


Figure 5: Style Files load times chart.

# of connections	Average load time (ms)
1	178.6
2	126.3
3	129.2
4	96.8
Chrome Browser	136.5

Figure 6: Style Files precise load times.

In figures 5 and 6, we notice a smaller decrease in load time when running parallel TCP connections. This is most likely due to the fact that CSS style files are already quite small in size and there is not much to be gained by parallelizing. The largest load time decrease happens when each CSS file is loaded through its own TCP parallel connection, which could be justified considering the small size of the files and the near 50% decrease in load time. Furthermore, our app once again outperforms the Chrome browser.

3.4 Style Files and Images

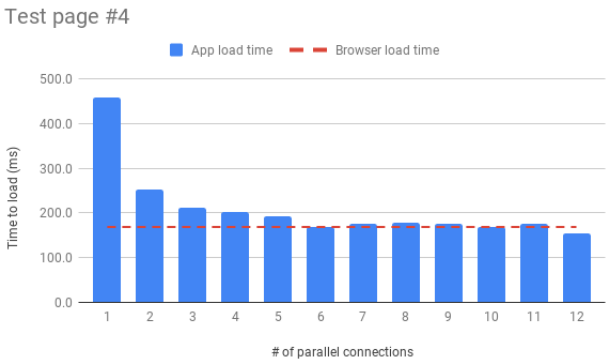


Figure 7: CSS and Images load times chart.

# of connections	Average load time (ms)
1	459.6
2	252.3
3	212.9
4	202.3
5	192.5
6	169.2
7	174.9
8	177.6
9	175.2
10	168.9
11	175.1
12	154.1
Chrome Browser	169.0

Figure 8: CSS and Images precise load times.

Lastly, we examine the results from loading our most diverse webpage, with a variety of CSS files and images. The data from figures 7 and 8 remarkably corroborates the conjecture we made previously - opening half as many parallel TCP connections as

there are resources results in near optimal performance. There is little to be gained in terms of load time when we open any more connections that that, and we could even risk a slower load time by doing so! Thus, we establish our formula through empirical testing as follows:

$$N = \frac{R}{2}$$

Where N is the number of TCP parallel connections being calculated, and R is the number of downloadable resources present on the page.

The following table shows the load time comparison between our finished app and the Chrome browser.

Test Page	App Average Load Time (ms)	Chrome Browser Average Load Time (ms)
Large Images	1925.7	2025
Moderate Images	404.6	419.5
CSS Style Files	133.6	140
CSS Files and Images	170.3	186

Figure 9: App vs Browser.

The results in the table were achieved through the same testing method as above. We ran four trials on every test page and averaged the resulting load times. It is clear that our app outperforms the Chrome browser in every test case.

4 Related and Future Work

4.1 Related Work

The use of parallel connections with TCP to improve performance has been previously studied. The effect of fetching different portions of a file through parallel connections to different servers is described in a 2002 paper by Rodriguez and Biersack [10]. Moreover, studies on various kinds of performance bugs in Google Chrome and other browsers have been conducted [6, 12].

There are also questions online about how to increase (or maximize) the number of parallel connections Chrome can have, and if there is a setting for this. We were not able to find such a setting [8, 9].

4.2 Future Work

While implementing the app, some ideas arose for the potential optimizations and improvements. The parallelization of resources using TCP connections could be performed even better by paying attention to the servers from which the various resources are loaded. For example, if two resources are loaded from the same server, we could keep the existing TCP connection to that server alive until both resources have been loaded. This would save some time on the creation and setup of a TCP connection. Similarly, we could reuse connections we have already created by connecting them to a different server, instead of creating a new connection every time. Another potential optimization would be to consider the size of the resources before deciding which connection they should be received through. Furthermore, it might be possible to break up a single large resource into several smaller parts and utilize the power of parallel connections to load even just one hefty resource.

5 Conclusions

Despite the limitations of Chrome's API, and the barriers to using TCP sockets API for extensions, we were able to pick the ideal number of parallel connections based on our experiments. Indeed, it slightly outperforms Chrome on all 4 trials for various data sizes, by using a formula that calculates the optimal number of TCP parallel connections to run to improve browser performance.

Google Chrome's optimization techniques are hard to outperform, but our app enables us to control some of the parameters that affect browser performance. While we were able to bypass some of the restrictions our app encountered to slightly outperform the Chrome browser, it leaves open the larger question of how well the app could perform if the formula is perfected in a less restrictive browser, or whether a browser API is the right medium to do this sort of task at all.

The source code for our app can be found at https://github.com/joom/connection-booster.

References

[1] S. Elan. How to communicate with webview in chrome app? https://stackoverflow.com/a/30375716/2016295. Accessed: 2019-01-14.

[2] Google. <webview> tag. https://developer.chrome.com/apps/tags/webview. Accessed: 2019-01-14.

[3] I. Grigorik. High performance networking in google chrome. https://www.igvita.com/posa/high-performance-networking-in-google-chrome/. Accessed: 2018-11-09.

[4] P. Irish. Chrome extension: Permissions to manipulate content of data uris. https://stackoverflow.com/a/48814186/2016295. Accessed: 2019-01-15.

[5] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[6] S. Lal and A. Sureka. Comparison of seven bug report types: A case-study of google chrome browser project. In *2012 19th Asia-Pacific Software Engineering Conference*, volume 1, pages 517–526, Dec 2012. doi: 10.1109/APSEC.2012.54.

[7] MDN web docs. Data uris. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs. Accessed: 2019-01-14.

[8] ProWebmasters. How can i tell chrome to use all 6 concurrent connections asap? https://webmasters.stackexchange.com/questions/90301/how-can-i-tell-chrome-to-use-all-6-concurrent-connections-asap. Accessed: 2019-01-15.

[9] Quora. What is the rationale behind setting the maximum number of connections per hostname to 6 for most of the browsers? https://www.quora.com/What-is-the-rationale-behind-setting-the-maximum-number-of-connections-per-hostname-to-6-for-most-of-the-browsers. Accessed: 2019-01-15.

[10] P. Rodriguez and E. W. Biersack. Dynamic parallel access to replicated content in the internet. *IEEE/ACM Transactions on Networking (TON)*, 10 (4):455–465, 2002.

[11] P. Technology. Diffusion cloud 6.1.3 user manual - browser connection limitations. https://developer.pushtechnology.com/cloud/latest/manual/html/designguide/solution/support/connection_limitations.html. Accessed: 2018-11-09.

[12] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 199–208, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1761-0. URL http://dl.acm.org/citation.cfm?id=2664446.2664477.