A Faster, More Efficient Internet Browser

Design Report

Aedan Cullen

4/2/2014

Period D

Purpose

I have created a faster, more efficient Internet browser (which I call Lightning) to ensure that as the Internet continues to grow, web pages always load fast and are responsive. Current browsers work well for what they are intended to do, but Lightning takes advantage of the unused RAM in modern computers to decrease page loading times.

Possible Solutions

While developing my web browser, I considered three possible techniques to use for increasing page loading speed. I decided on the one which I thought would provide the best results and the most improvement over existing web browsers.

Solution 1: Proxy Server with URL-Based Link Ranking

My first idea was to use a proxy server to cache pages that users would be likely to visit. When requests were made, the proxy would monitor the URLs and calculate the pages users would be most likely to visit based on their previous actions. Then, these pages, along with their corresponding resources, would be downloaded to the proxy server and stored there for later use. Finally, if the user's web browser requested these pages again, the proxy server would have them ready for fast downloading to the browser.

There are several advantages and disadvantages to this approach. The first advantage is ease of implementation. It would be simple to setup a proxy server for a network, and installing software on individual computers would not be required. However, if multiple computers on the same network were using the same proxy server, there would be no way to tell which computer was making which requests. As a result, it would be impossible to monitor individual users' actions. Also, this method would probably not improve on loading times very much, because pages would still have to be downloaded to the user's web browser. The only

improvement would be that the pages' source would be a server closer on the Internet to the receiving computer. So, if the bottleneck slowing down Internet speeds was on a local network, no improvement would be made. Therefore, this is not the best solution.

Solution 2: Custom Web Browser with Style-Based Link Ranking

My second idea was a custom browser. It would monitor the pages users visited and evaluate rankings of the links on those pages, based on their style. The browser would then calculate the links that users would be most likely to visit. For example, bold, colorful, or animated links would be more likely to be visited. Then, these links would be prerendered in the background, providing faster page loading times when the links were actually clicked. This is a method that would improve on the current implementation of link prerendering in some browsers, which relies on the page to provide hints in <link> tags or HTTP headers. Instead, my browser would interpret the page intelligently instead of relying on website developers to provide the information.

This method has many advantages, and almost no disadvantages. It does not rely on web pages to provide information about what links to prerender. This would allow the browser to intelligently predict the user's actions on any page. However, a disadvantage is that this method also relies on information in websites. Even though not directly, the information being used to rank the links would be coming from the designer of the page. This is something I do not want, as a better source of this type of information is the user. This leads to my third solution idea, which learns about the user and manages multiple rendering widgets efficiently.

Solution 3: Custom Browser with User-based Link Ranking

My final and chosen idea was a custom browser similar to the one in Solution 2, but with a different method of prerendering. Instead of relying on the designers of pages to provide information about the importance of links, this method would actually learn about the user as

they clicked links. It would maintain a database that would keep track of how many times the user had clicked certain links. You can read more about this in the section titled “Browser Design”. In short, the more times a user had clicked on a link in the past, the greater the chance that they would click on the link again.

I chose this method because it does not rely on anything other than the user to provide information about the importance of links. It also varies its rankings based on individual users’ favorite pages and things to do on the Internet. This method can create a browser that is fast, and learns about users to predict their actions. Clearly this method is better than both of the others.

Best Solution

My final solution was Solution 3, described above. I decided to use the Python 3 programming language, with PyQt for a GUI and WebKit for page rendering. I also use the BeautifulSoup module to parse HTML data and extract anchor (“*<a>*”) tags. PyQt provides a great way to create the GUI in this case, since it has support for stacked widgets (*QStackedWidget*) and a special *QWebView* widget for use with WebKit. It also makes available plenty of signals from WebKit, for various purposes such as detecting when a page has finished loading, or capturing a link-clicked event. Also, I use the *time* module from Python’s standard library for collecting page loading times using the *time.clock()* method.

Browser Design Note: Diagrams are provided at the end of the report

Lightning uses a GUI similar to that of many other web browsers. It has a toolbar across the top (using QToolBar) with buttons (QPushButton) on the left for Back, Reload, and Home, and a URL bar (QTextEntry) in the center. To the right of the URL bar is a drop-down menu

button (QPushButton with a menu attached) with a gear icon. When clicked, this shows a context menu with various settings and actions. In the center of the browser window is a QStackedWidget that contains the various QWebViews Lightning will be using to display pages. From now on I will sometimes refer to the QStackedWidget in the center of Lightning as the "stack".

To decrease the loading times of pages, Lightning uses two entirely new ideas which I invented myself. The first is intelligent prerendering. The technique of link prerendering has already been implemented in some web browsers, but my version improves on it. Normally, a browser will only prerender links which the webpage has instructed the browser to prerender. If a page does not provide any information about what links should be prerendered, then the browser will not prerender any links. Also, for different users and on different computers, the page/pages that are prerendered will always be the same. Lightning, however, uses an algorithm to calculate which links a user will be most likely to visit on each page. It does this by learning about users based on their previous actions.

The idea behind this is simple: The links that a user has clicked on *most* in the past are the ones the user is most likely to click on again in the future. Using this idea, the browser can prerender links intelligently based on what the user has done in the past without relying on web pages to provide information. Lightning uses a new algorithm to learn about users and predict which links they will be most likely to click on in the future.

The algorithm uses a system of ranks to choose the links to prerender that have been clicked on the most in the past. Links with higher ranks have been clicked on the most, and are therefore most likely to be clicked on in the future. Links with a lower rank have not been clicked on as often, and are not as likely to be clicked on. Each link's rank starts at 0. When a link is clicked, its rank is incremented by 1. At the same time, the ranks of all the other links on the

page are decremented by the quotient of *1 divided by the number of non-clicked links*. This way, the average rank of all the links on any given page stays at 0.

When a page is then visited in the browser, the URL of the highest-ranked link is retrieved from the database (the one the user is most likely to click) and then prerendered in the background. For example, imagine there are only three links on a page, and none of them have been clicked yet:

Link A: 0

Link B: 0

Link C: 0

All of the ranks are zero. Now, if a user clicks on link A:

Link A: 1

Link B: -0.5

Link C: -0.5

Now link A has the highest rank, because it has been clicked on the most and is therefore most likely to be clicked on in the future. The average rank is still 0.

Now, link B is clicked:

Link A: 0.5

Link B: 0.5

Link C: -1

Now, Link A and Link B have the same rank, because they both have been clicked on once. You can see that depending on which links are clicked, the rankings adjust accordingly to predict the user's future actions. In this case, if link C was clicked once also, the rankings would even out once more, at 0, 0, and 0, because all links would have been clicked the same number of times.

Lightning also needs link ranks to persist even if the browser is closed or the computer is

restarted. Otherwise, information about a user's actions would be discarded when the browser was closed. This is done using a DBM database managed using Python's *shelve* module. The database stores visited page URLs in its keys and dictionaries as its values. Inside the dictionary for any given page, the keys are the HREF attributes of the links on the pages, and the values are the links' rankings. When the browser is started, this database is read from and the information is loaded into memory. When the browser is closed, new ranks are written to the database before the Python process exits. Since the browser creates a new rank database on startup if one does not already exist, ranks can be reset by simply deleting the database.

See Diagram 1: Database Structure

This ranking algorithm lets the browser predict the user's future actions accurately, without relying on web pages to provide prerendering hints. After Lightning has determined which link on a page should be prerendered, the page is then loaded completely in the background, without the user even noticing. For this a PyQt QStackedWidget is used. The QStackedWidget maintains a stack of QWebView widgets, and when a page needs to be prerendered, a new WebView is simply added to the stack and instructed to begin the prerender. The user never sees it, and if the prerendered link is then clicked, the stack's index can be set to reference the QWebView containing the prerendered page. Since the only thing happening then is the changing of the QStackedWidget, the result is a page that loads very, very fast, with times usually quicker than 10ms in my tests. Of course, this happens only when the prerendered link is clicked by the user. At first it may seem that this chance is small, but the ranking algorithm is continuously learning about the user every time they click a link on the page. As a result, the chance that the correct page has been prerendered is much higher in Lightning than it would be in an ordinary browser where no prediction/ranking algorithm is used.

See Diagram 2: Intelligent Prerendering

The second of my improvements is a new way to manage pages the user visits. In a normal web browser, when a page is visited, it is downloaded and displayed on-screen. But when another page needs to be loaded, either because a link was clicked, or because a new URL was entered in the URL bar, or for another reason, the first page is discarded so that the next can be loaded. Although a browser may keep elements of the page in its cache, such as images or scripts, the entire page is not kept in memory. It is downloaded and rendered again when needed. Although the caching of resources definitely improves performance when users return to pages they have already visited in the same browser session, speeds could furthermore be improved here if the browser kept already-visited pages stored in the computer's RAM for quick access. This is something that Lightning does natively. When a new page needs to be loaded, a new WebView is created in the QStackedWidget, and its URL is set to the page that needs to be loaded. Then, the index of the stacked widget is set to point to the new WebView widget, and the old page stays in the stack. This way, if the browser needs to load the page again, it will find it already in the stack, and be able to display it very quickly. This is much more efficient than a normal web browser, in which the page would have just been discarded and re-downloaded. So, if the user revisits a page in Lightning that they have already visited during the same browsing session, the page will be found already in the stack. This ends up happening very often if a user is revisiting pages often in the same browser session. Loading times when this happens in Lightning are about the same as cases where the page is prerendered, because the browser is completing the same action (just changing the index of the QStackedWidget). If a user keeps the browser open and visits many pages, the stack will collect these pages and store them until the browser window is closed.

See Diagram 3: Efficient Page Managment

When populating a QStackedWidget with many pages, a PyQt application's RAM usage

goes up because each page of the QStackedWidget must be stored, even the ones that are not visible. Since Lightning is storing many complete web pages in its stacked widget all at once, the amount of RAM used is even greater. The exact amount of memory used, of course, depends on how many pages are actually in the stack and how many resources (such as images) each page is using. As a result, Lightning sometimes uses more RAM than other popular web browsers such as Google Chrome. This is the tradeoff that happens from storing many pages in order to improve loading times. However, on modern computers that have 4GB or more RAM, this is not really an issue. Since most computers are never even close to using all of their RAM, this extra memory space can be used for optimizing speeds.

Modifications

During software development, many, many modifications are made. As a result, there are too many revisions and changes to Lightning's source code to list here, but I will provide a changelog of major additions and improvements.

Version 0.9.1 (12/30/13)

-Started working on basic functions

Version 0.9.2 (1/2/14)

-Added prerendering stack

-Began work on ranking algorithm

Version 0.9.3 (1/8/14)

-Added simple prerendering

Version 0.9.4 (1/14/14)

-Added database management functions

Version 0.9.5 (**1/20/14)**

-Finished ranking algorithm

-Wrote Python implementation of rank algorithm

Version 0.9.6 (2/6/14)

-Added link-ranking algorithm to browser

Version 0.9.7 (2/11/14)

-Split page-display code into two sections for simplicity

Version 0.9.8 (2/20/14)

-Added link-clicked event handler

-Added rank-updating function

Version 0.9.9 (3/11/14)

-Fixed KeyError related to non-existent database entry

Version 1.0.0 (3/20/14)

-Prevented loadingDone event from firing multiple times by using loadProgress instead and waiting for it to reach 100

Testing and Results

To determine if my web browser was actually faster than existing browsers, I tested it against the latest version of Google Chrome, which is extremely popular and most likely the fastest browser around today. My tests involved first loading the web browser's homepage, which I set in Lightning to be the same as in Google Chrome. (http://www.google.com) Next, I loaded a separate page, on which I clicked a link, then returned to the first page. I then clicked on the link again. Finally, I closed the browser window, re-opened it, went back to the page I had been on, and clicked the same link yet another time. This series of actions provides

opportunities for both of Lightning's improvements to run, and also allows Google Chrome's caching techniques to be used. I performed all steps on both browsers at the same time and on the same Internet connection with the same computer. The computer I used has an Intel Core i3 - 350M CPU, 4GB RAM, and Windows 7 Home Premium on a 500GB 7,200rpm HDD. I used a Verizon 1Mbps DSL Internet connection. Finally, I also made sure that neither browser had information stored in its cache before each test began, and I reset Lightning's rank database before each test. Below are a series of tables that show each of my tests, and the times I collected. All times for Lightning are measured in its Activity Log toolbar which uses Python's time.clock() method, and times for Google Chrome are measured using the Network section of the browser's Developer Tools window.

Test 1: Google Accounts Login (3/18/14)

Action	**Load Time in Lightning (seconds)**	**Load Time in Google Chrome (seconds)**
1. Load homepage (www.google.com)	3.1108 (non-prerendered)	4.83
2. Link clicked: Sign In	1.7414 (non-prerendered)	3.14
3. Return to google.com (address bar)	0.9926 (non-prerendered)	1.23
4. Link clicked: Sign In	0.0047 (Already in stack)	0.334
5. Close browser and re-open (back to google.com)	N/A	N/A
6. Link clicked: Sign In	0.1114 (Prerendered)	0.977
Total:	**5.9609 seconds**	**10.511 seconds**

Total time difference: **4.5501** seconds faster

Test 2: Google Maps (3/17/14)

Action	**Load Time in Lightning (seconds)**	**Load Time in Google Chrome (seconds)**
1. Load homepage (www.google.com)	3.6396 (non-prerendered)	8.40
2. Link clicked: Maps	13.8176 (non-prerendered)	8.36
3. Return to google.com (address bar)	1.0999 (non-prerendered)	1.33
4. Link clicked: Maps	0.0074 (Already in stack)	4.32
5. Close browser and re-open (back to google.com)	N/A	N/A
6. Link clicked: Maps	0.0068 (Prerendered)	4.19
Total:	**18.5713 seconds**	**26.6 seconds**

Total time difference: **8.0287** seconds faster

Test 3: Mount Snow Events page (3/23/14)

Action	**Load Time in Lightning (seconds)**	**Load Time in Google Chrome (seconds)**
1. Load www.mountsnow.com	25.4966 (non-prerendered)	44.10
2. Link clicked: Events	5.1768 (non-prerendered)	6.33
3. Return to mountsnow.com (address bar)	7.4576 (non-prerendered)	17.02
4. Link clicked: Events	0.0033 (Already in stack)	3.91
5. Close browser and re-open (back to mountsnow.com)	N/A	N/A
6. Link clicked: Events	0.018 (Prerendered)	3.85
Total:	**38.1523 seconds**	**75.21 seconds**

Total time difference: **37.0577** seconds faster

Test 4: Mount Snow Explore page (3/28/14)

Action	**Load Time in Lightning (seconds)**	**Load Time in Google Chrome (seconds)**
1. Load www.mountsnow.com	22.363 (non-prerendered)	31.02
2. Link clicked: Explore	5.773 (non-prerendered)	5.18
3. Return to mountsnow.com (address bar)	5.3585 (non-prerendered)	7.53
4. Link clicked: Explore	0.003 (Already in stack)	3.28
5. Close browser and re-open (back to mountsnow.com	N/A	N/A
6. Link clicked: Explore	0.0144 (Prerendered)	3.27
Total:	**33.5119 seconds**	**50.28 seconds**

Total time difference: **16.7681** seconds faster

Test 5: Hopkins Music (3/30/14)

Action	**Load Time in Lightning (seconds)**	**Load Time in Google Chrome (seconds)**
1. Load hadleyhopkins.vt-s.net/Pages /index	4.1042 (non-prerendered)	4.86
2. Link clicked: Hopkins Music	1.764 (non-prerendered)	1.17
3. Return to hadleyhopkins.vt-s.net/Pages /index (using address bar)	2.302 (non-prerendered)	3.14

4. Link clicked: Hopkins Music	0.0042 (Already in stack)	0.633
5. Close browser and re-open (back to hadleyhopkins.vt-s.net/Pages/index)	N/A	N/A
6. Link clicked: Hopkins Music	0.0051 (Prerendered)	0.719
Total:	**8.1795 seconds**	**10.522 seconds**

Total time difference: **2.3425** seconds faster

Test 6: Campmor (4/1/14)

Action	**Load Time in Lightning (seconds)**	**Load Time in Google Chrome (seconds)**
1. Load www.campmor.com	17.3999 (non-prerendered)	17.01
2. Link clicked: Gear	2.2244 (non-prerendered)	1.91
3. Return to campmor.com (address bar)		
4. Link clicked: Gear	0.003 (Already in stack)	1.85
5. Close browser and re-open (back to campmor.com)	N/A	N/A
6. Link clicked: Gear	0.0041 (Prerendered)	1.95
Total:	**19.6314 seconds**	**22.72 seconds**

Total time difference: **3.0886** seconds faster

Conclusion

Lightning performed faster than Google Chrome in all of these tests. Sometimes, in

situations where Lightning's improvements are not running (non-prerendered pages) it still has a faster time. This is because Lightning currently does not have all the features that Google Chrome does. As a result, pages can sometimes load faster just because the browser is simpler and not a lot of features need to be managed at the same time pages are loading. This shows that there is also an advantage to a simple browser over a very complicated one.

In situations where the improvements were in action (prerendered pages and already-in-stack pages) Lightning's times were all faster than 200 milliseconds. This was achieved because the browser only had to change the index of the QStackedWidget in order for the page to be displayed. In all of the tests, Lightning beat Google Chrome by several seconds in total. Although Chrome was a tiny bit faster in a few ordinary page loads, Lightning's super-fast "already in stack" and "prerendered" times showed it was superior overall.

Altogether, Lightning was a success and is faster than Google Chrome. It does not yet have all the features of current Internet browsers, but definitely has an edge in speed due to its two improvements. A faster, more efficient Internet browser has no doubt been created, and will hopefully help counteract the slow loading speeds of certain web pages at times.