Assignment 3:

Feature Proposal: Enhanced Chrome Sync

30 November 2018

Course Code: CISC 322
Instructor: Dr. Ahmed Hassan
TA: Hongkai Chen

Cameron Raymond – 15cjkr@queensu.ca
Brendan Russell – 15br18@queensu.ca
Brenden Forbes – 15bpf@queens.ca
Christopher Molloy – chris.molloy@queensu.ca
Michael Wrana – 16mmw@queensu.ca
Ross Hill – ross.hill@queensu.ca

Table of Contents

Abstract

In 2012, Google introduced a feature for Chrome that keeps track of what tabs the user has open and syncs them across the user's devices (Keeping Tabs on Your Tabs). For example, if someone open a news article on their desktop, when they open Chrome on their mobile device the news article would be open. Google achieves this through "Google Chrome Sync", which also synchronizes data like passwords, history, and bookmarks (Sync). For our feature, we are making it so that synchronized tabs keep track of webpage progress - that is, how far into the content of the webpage the user has gotten. For now, webpage progress data is comprised of how far the user has scrolled down the webpage and how far into the webpage's embedded videos the user has watched. Our team was motivated to implement this feature because it would allow Chromium users to transfer their work across devices more efficiently.

Introduction and Overview

After investigating the concrete architecture of Chromium, we are now able to see how our feature might be implemented. Through examining Chromium's architecture, one finds that Chromium's sync feature is currently housed in the Browser Engine. Chrome Sync interacts with Network in order to communicate with Google's servers. Chrome Sync stores updated data locally with Data Persistence.

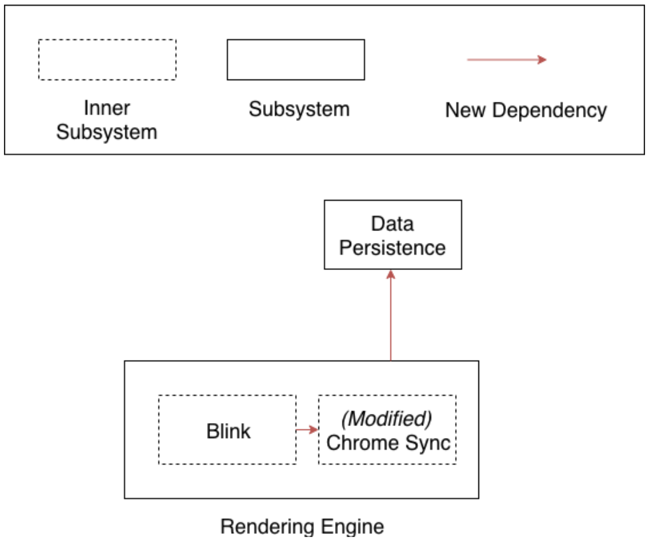
A challenge with syncing webpage progress across browsers is that Chrome Sync will need access to information on the state of the webpage, namely how far the user has scrolled and how much time is left on any embedded videos. Chrome Sync needs to communicate with Rendering Engine to get this webpage state information. In order to facilitate this, we explore moving the Chrome Sync from Browser Engine to Rendering Engine, but ultimately, we decide that Chrome Sync belongs in Browser Engine due to the kind of data it manages. This approach also fulfills a variety of non-functional requirements like maintainability and manageability better than the alternative implementation.

Implementation

Alternate Implementation

When deriving possible implementations, one possible solution that stood out was moving the modified Chrome Sync to the Rendering Engine. As shown in the second sequence diagram, the Rendering Engine takes in the webpages' page data and page state, and then builds the modified DOM that is then returned to the user. Having Chrome Sync in the Rendering Engine would allow for this to be captured in one subsystem, but possibly reduce the performance of the subsystem, which is an important NFR. As well, the Rendering Engine communicates directly with the JavaScript engine (V8). If this feature was to be modified in the future to capture the state of the executing JavaScript, then this communication would important.

While this makes moving Chrome Sync to the Rendering Engine an appealing first choice there are multiple problems. The main issue is that Chrome Sync involves communication with the data persistence subsystem. This means that having the Rendering Engine handle Chrome Sync would involve extra dependencies between the Rendering Engine and data persistence subsystems, as shown in the diagram below. This extra complexity would reduce the reliability of the system. As well, the Rendering Engine as of now has the sole purpose of creating renderers for Browser Engine. Shoehorning the syncing capabilities into this subsystem reduces its overall cohesion. For these reasons, possible bugs that are introduced become much harder to find and fix, affecting the overall systems maintainability and reliability. Our team decided that for this specific feature, the benefits of moving Chrome Sync to the Rendering Engine violated the core principles of the Object-Oriented architecture style and continued to look for possible solutions.



Decided Implementation

In the decided implementation of our feature, we keep Chrome Sync within Browser Engine. Chrome Sync is the subsystem that keeps data synced between a user's Chromium instances. If Chromium only synchronized data related to how the webpage gets displayed, putting Chrome Sync within Rendering Engine may have made sense. However, the information that is synced includes data unrelated to webpage rendering like passwords and browser history (Sync) and making the Rendering Engine manage data unrelated to its purpose of rendering webpages would decrease its cohesion.

Though Browser Engine still handles Chrome Sync, Rendering Engine is used in our decided implementation to retrieve the webpage state information to be synched. Periodically, Browser Engine's Chrome Sync subsystem requests a webpage progress update from Blink. The update includes information on how far down the page the user has scrolled and the time remaining on videos embedded in the page. If the change in webpage progress is significant enough (for example, the user has scrolled more than a few pixels), Browser Engine caches the updated webpage progress with Data Persistence and sends it to Google's sync servers to be synchronized across the user's devices.

When the user opens another Chromium browser instance associated with their Google account, the Browser Engine's Chrome Sync subsystem requests updates on synced tabs. If webpage progress on a synced tab has changed, the updated webpage progress (scroll distance and video progress) is stored with Data Persistence and is passed to the Rendering Engine.

SAAM Analysis

Stakeholders are anyone who has an interest in, or concerns relative to, a system. The two stakeholders of this implementation are users, and Chrome developers. A Non-Functional Requirement (NFR) is a description of how well a system performs its functions. Both stakeholders depend on certain non-functional requirements for the implementation to be a success.

Stakeholder	Non-Functional Requirement
User	● Performance ● Accuracy ● Portability
Chrome Developer	● Maintainability/Testability ● Manageability ● Evolvability

The NFRs of the user are performance, accuracy, and portability. When comparing which of the two subsystems the feature should be implemented in, the most important NFR is Performance. If the feature is implemented in the Rendering Engine, then each of the renderers would be tasked with recording and storing their own page data. Each of the renderers already handle a lot of tasks, so adding on the job of recording page data will lower the performance of each renderer, and the Rendering Engine as a whole. If the feature is implemented in the Browser Engine there will be one central space for all page data to be recorded and stored. This lowers the total effect the feature will have on the system. The accuracy and portability of the implementation is not dependent on the chosen subsystem.

For this implementation, Chrome developers have two NFRs: maintainability and manageability. Both features focus on testing to ensure that nothing is wrong with the implementation and ensuring that there is a system in place for the developers, if something does go wrong. Neither of these requirements would be significantly affected by which subsystem the feature is added to. But again, if the feature is added to the Browser Engine, there will be one central place for all the data to be collected, and due to Rendering Engine creating and removing renderers this may cause more problems when considering these NFRs. Additionally, the evolvability of the system is unaffected by the chosen subsystem.

From analyzing the NFRs of the system it was concluded that the feature will be implemented in the Browser Engine. Furthermore, implementing the feature in the Browser Engine will lower the cohesion of the system, but implementing in the Rendering Engine would lower cohesion and raise coupling, due to the feature depending on the Data Persistence subsystem.

Interaction with Other Features

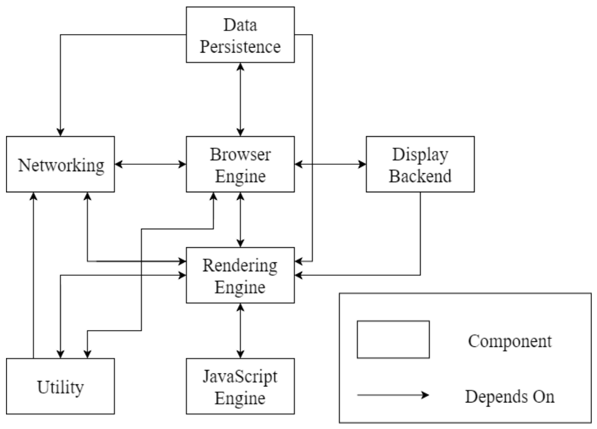
Impacted Subsystems

The main impacted subsystems are the Browser Engine, Rendering Engine and Data Persistence. We decided to store our feature in the Browser Engine, and the Rendering Engine is responsible for a lot of its implementation.

In the process of sending updated web page progress, the Rendering Engine will update the Browser Engine when the state of the webpage is sufficiently changed by scroll distance or video progress. Generally, when there is a scroll, the Rendering Engine will update its state. The Browser Engine will then push the updated state to Google sync servers. Google Sync handles history, bookmarks, and settings between devices and will also be responsible for our feature.

The current page state will be stored in data persistence. To receive the updated page on another device, the Browser Engine receives an update from the google sync servers. This is stored in data persistence and sent to blink in the Rendering Engine. Blink then updates the page to its new state.

This feature will not have a significant impact on our network subsystem. Ultimately, this does not change our conceptual architecture in any way as the browser and Rendering Engine remain interdependent. Interactions with data persistence also will not change as they are already interdependent.



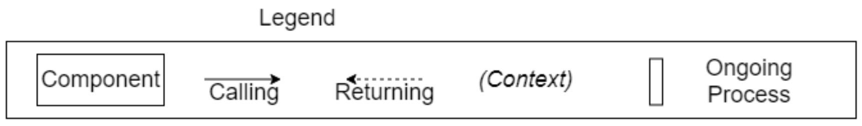
Flagged Files/Directories

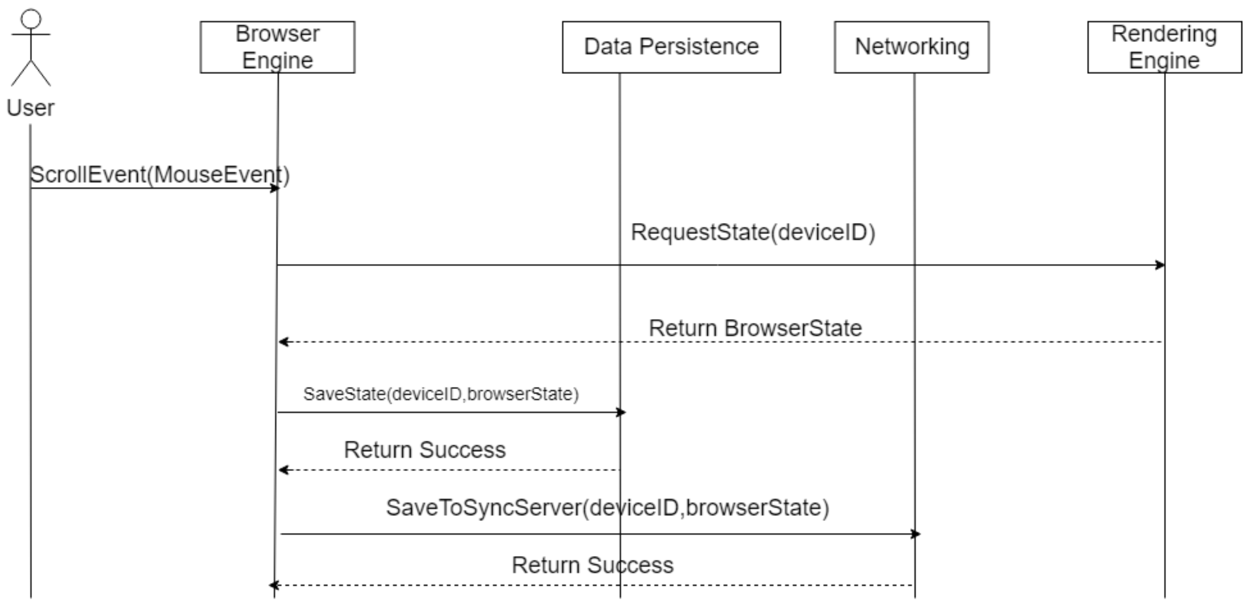
The sync folder in our Browser Engine will need to be expanded to accommodate the new feature. Right now, it is comprised of files responsible for syncing bookmarks, history, chrome profiles and settings. We would need additional files in this directory to define what constitutes a webpage, syncing the webpage and additional test cases within the test subdirectory. We will need to also make additional files in the Rendering Engine to track the webpage and video progress to push to the Browser Engine.

Google's framework already uses XMPP Google Talk servers which push when information is changed instead of polling periodically ("Sync", The Chromium Projects). When a change is registered in a Chrome client, an XMPP message is sent to other clients which tells them to sync. With this framework already in place, it should make it easier to implement and scale our feature.

Sequence Diagrams

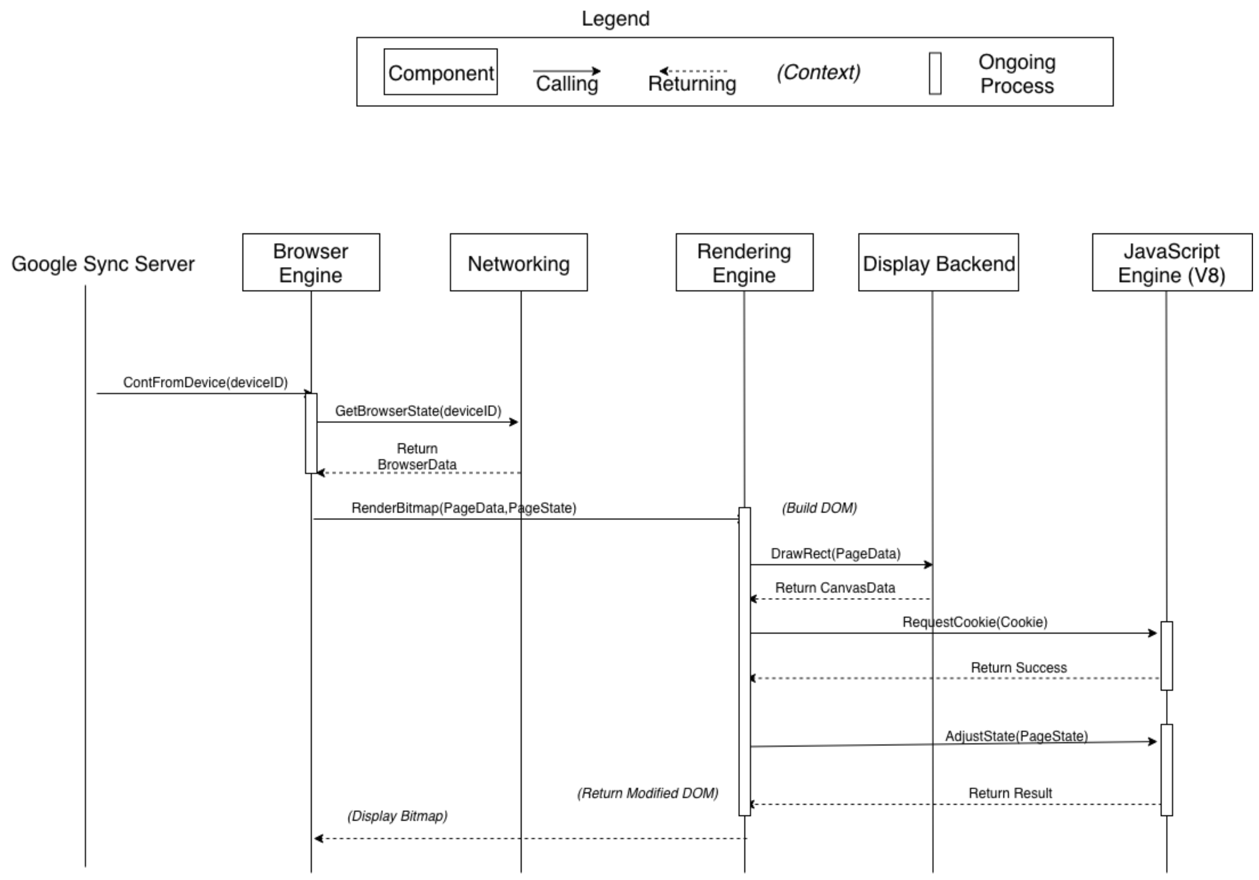
Sequence Diagram One - Page is scrolled as user is browsing





This sequence diagram describes the series of events that occurs for a page state to be saved when a user is browsing. This is so that if the page is to be closed at any point, the most recent state has been saved. This occurs any time the page is scrolled significantly, shown in a scroll event sent by the user. This causes the Browser Engine to request the current page state based off the device ID from the Rendering Engine. This state is then saved locally in data persistence and sent to the networking subsystem to be saved to the Google Sync servers.

Sequence Diagram Two - Continuing from a Device with One Tab Open



The webpage request begins with Google Sync Server communicating to the Browser Engine that a page state can be pulled from a specific device; this would either be the default device that the user set to sync with, or the device chosen when the "What device would you like to continue from?" pop-up is shown. The Browser Engine handles this request and sends the deviceID to the networking component to request the stored page data for that specific device. After page data is received by the Browser Engine, it is sent to the Rendering Engine. The page data is rendered in the display backend, while the PageState is sent to V8 (cs.chromium.org). Since a cookie is required, Blink forwards a cookie to the JS engine which interprets the JavaScript and modifies the DOM to continue from the point in the page as specified by the PageState parameter (V8 Project Authors). Once rendering is complete, the modified DOM is sent to the Browser Engine and displayed through the UI to the user.

Test Cases

After this feature is completed, we need a way to ensure that the new system works as intended. Furthermore, any other parts of Chrome should work exactly the way they did before any changes were made. Due to the size and complexity of Chrome, complete functional verification would be extraordinarily difficult. Listed here are four possible test cases that can be used to verify the plugin’s basic functionality and ensure Chrome is still working correctly.

The first basic test to perform is simply loading webpages in Chrome normally. This will verify that all the basic features are working correctly. A worst-case scenario for adding a feature is Chrome being completely broken, which might cause users to switch to another web browser. While it is highly unlikely this could result from implementing our feature, it does modify the browser engine, and anything is possible considering how central that subsystem is.

The second and third tests performed are to verify our feature works correctly. This involves closing a tab while browsing, then reopening it on the same device. If this part of the feature works correctly, the tab should load to the same part of the webpage when it was closed. The next test also begins by closing a tab in chrome. However, the browser should be reopened on a different device and still sync to the correct position on the webpage.

The last test is more complicated and designed to simulate a real-world use of this feature in Chrome. First, the user will login to their Google account, open a specific webpage, then logout of their account. Next, they will login to their account on the initial device and a separate device. If our feature works as intended, when the tab reopens, it should automatically go to the correct part of the webpage on both devices.

Risks & Limitations

There are three risks for this implementation, the first two: change and annoyance are quite similar. Both relate to how the users respond to the feature being implemented. Whenever a new feature is added to software systems there is usually some blowback from the users. People do not like change, and there is not a lot that can be done to evade this risk other than making the feature as unobtrusive as possible. Constantly having a popup open whenever a user opens chrome on a device may be annoying if they do not use the feature frequently. A fix for this is adding an option to turn off the implemented feature in the settings. The third and biggest risk of this feature is security, currently the only protection comes from the Google account connected to each device. If a user’s Google account were to be compromised, then there is nothing stopping the offender from seeing exactly what the victim is doing online.

The limitation of the feature currently is how it will deal with JavaScript. The current implementation records and stores how far down a page, or how long into a video the user is, and sends that information off to the next device the user operates. This does not consider any JavaScript that may be running on that webpage when the last recording is taken.

The largest limitation for implementing this feature was the size of Chrome. Chrome is a massive system with over 5 million lines of code and doing basic things like searching for the correct flagged files and directories became quite the challenge. The biggest lesson that we will take away is how difficult it is to implement a feature as small as ours into a software system. We initially were not aware of how much work would have to be done to implement something that is not very complex.

Data Dictionary & Naming Conventions

AOT: Ahead-of-time (compilation)
AST: Abstract Syntax Tree - A tree structure denoting the semantic routes possible in source code
CSS: Cascading Style Sheets - A language for formatting HTML components
Cache: A small amount of fast, expensive memory
DOM: Document Object Model - A universal specification for laying out and giving access to HTML components
HTML: HyperText Markup Language - An encoding language used to format a static web page layout
SQL: Structured Query Language - Standard language for database management and communication
HTTP: HyperText Transfer Protocol - defines how messages are formatted and transmitted on the World Wide Web
IPC: Inter-Process Communication - the communications channel created to deliver messages between processes spawned by Chrome
Ignition: Chrome V8's Baseline Compiler
TurboFan: Chrome V8's Optimising Compiler
JIT: Just-in-time (compilation)
URL: Uniform Resource Locator- reference to a web resource on a network and how to retrieve it
MLOC: Millions of Lines of Code
Hot Function: A function that is often called with the same argument types

References

Boyd, Ian. “How Does Google Chrome Store Passwords?” Super User, 2013, superuser.com/questions/146742/how-does-google-chrome-store-passwords.

Cs.chromium.org, 2013, cs.chromium.org/chromium/src/out/android-Debug/gen/components/autofill/content/common/mojo_types_java/generated_java/org/chromium/autofill/mojom/PasswordForm.java?type=cs&q=PasswordForm&sq=package%3Achromium&g=0&l=17.

V8 Project Authors. “V8 Documentation” V8 JavaScript Engine, https://www.v8.dev/docs.

Nickolov, Lachezar. How JavaScript works: Parsing, Abstract Syntax Trees (ASTs) + 5 tips on how to minimize parse time. https://blog.sessionstack.com/how-javascript-works-parsing-abstract-syntax-trees-asts-5-tips-on-how-to-minimize-parse-time-abfcf7e8a0c8

Kennedy, Niall. (2008). The Story Behind Google Chrome. https://www.niallkennedy.com/blog/2008/09/google-chrome.html.

“Multi-process Architecture.” The Chromium Projects, https://www.chromium.org/developers/design-documents/multi-process-architecture.

“Network Traffic Annotations.” Threading and Tasks in Chrome, https://chromium.googlesource.com/chromium/src/ /lkgr/docs/network_traffic_annotations.md.

“Sync.” The Chromium Projects, https://www.chromium.org/developers/design-documents/sync

“Keeping Tabs on Your Tabs.” *Google Chrome Blog*, 15 May 2012, https://chrome.googleblog.com/2012/05/keeping-tabs-on-your-tabs.html.