CISC 322

Assignment 1: Report

Google Chrome's Browser: Conceptual Architecture

Friday, October 19, 2018

Group: Bits...Please!

Emma Ritcey *15er21@queensu.ca*

Kate MacDonald *14km90@queensu.ca*

Brent Lommen *brent.lommen@queensu.ca*

Bronwyn Gemmill *14bvg1@queensu.ca*

Chantal Montgomery *15clm1@queensu.ca*

Samantha Katz *12sk93@queensu.ca*

Abstract

The Google Chrome browser was investigated to determine its conceptual architecture. After reading documentation online and analyzing reference web browser architectures, the high level conceptual architecture of Chrome was determined to be a layered style. Individual research was done before collaborating as a group to finalize our proposed architecture. The conceptual architecture was proposed to coincide with Chrome's four core principles (4 S's): simplicity, speed, security, and stability. In depth research was completed in the render and browser engine subsystems which had the architectures styles object oriented and layered, respectively. Using the proposed architecture, the process of a user logging in and Chrome saving the password, as well as Chrome rendering a web page using JavaScript were explored in more detail. To fully understand the Chrome browser, Chrome's concurrency model was investigated and determined to be a multi-process architecture that supports multi-threading. As well, team issues within Chrome and our own team were reported to support our derivation process and proposed architecture.

Table of Contents

Introduction

Ten years ago, Internet Explorer and Firefox were the most popular web browsers in the world, controlling 63.1% and 29.1% of the web browser market share in January 2008, respectively. The way the internet was being used however, had changed dramatically since web browsers had first been engineered. Web pages weren't just displaying text anymore; they were being used for entertainment such as gaming and streaming videos. They had become applications. Google wanted to evolve with the internet and recognized that they could create a brand-new platform that revolutionizes the way web browsers function, improving the experience for users.

This idea led to Google Chrome: an open-sourced web browser. It was designed completely from scratch, specifically to meet the increasing needs of internet users. Throughout development, the Chrome team focused on simplicity, speed, security, and stability, which they referred to as the 4 S's. Their goal was to improve the internet as a whole which was why they decided to make Chrome an open-sourced project. They have made design documents, architecture overviews, and more available so that they can share their development process with the community and help other developers improve their programs.

The success of Chrome is shown in the numbers. In September 2009, one year after its release, Chrome controlled 3.65% of web browser market share worldwide. Ten years since its release (September 2018), Chrome controlled well over half of the world web browser market share at 60.63%. This success can be largely credited to its well-designed conceptual architecture, as it creates a positive, secure experience for users that makes them want to continue to use their browser.

The goal of this report is to determine and analyze the conceptual architecture of Chrome. After studying reference architectures and online resources, we concluded that the conceptual architecture consists of 7 subsystems: User Interface, Memory, Network, Plugins, Browser Engine, Render, and JavaScript V8. After analyzing these subsystems and determining the dependencies between them, we concluded that the conceptual architecture of Chrome is layered at a high-level. Looking at lower levels within subsystems, the object-oriented architecture style is also used. However, what separates Chrome from all other web browsers is its use of a multi-process architecture. A multi-process architecture allows the Browser Engine to create a separate process for each new tab or plugin that is running. This means that if one process crashes, the other processes will be safe and the tabs will continue to function.

Along with describing the conceptual architecture and the interactions between the subsystems, we will provide further detail into the architecture within the Browser Engine and Render. Following that, we will describe two use cases and how they interact with the various parts of the architecture to perform successfully. Then we will discuss various

limitations we experienced as a team and lessons we have learned, as well as issues the Chrome team has had while developing and refining Chrome.

Derivation Process

Formulating the conceptual architecture of Chrome required a significant amount of research to fully understand Chrome and the way it was designed. The Chromium Blog along with a comic book written by the Google Chrome Team entitled "Behind the Open Source Browser Project" proved to be helpful sources in the initial stages of research. They exposed us to the development team's goals of Chrome being fast, secure, stable, and simple, and the reasons behind their decision to make Chrome open-source. Once we were familiar with how and why Chrome was designed, we began exploring the Chromium Project Design Documents. These design documents contained much more technical information about the architecture of Chrome, and although they focused on lower level systems than we were interested in, they still proved to be very useful in coming up with our final conceptual architecture.

Using the previously mentioned sources along with other online resources and a reference architecture for web browsers, each member of the group formulated their own conceptual architecture. Then, we came together and shared the architectures, discussing the pros and cons of each. Each architecture differed slightly in which subsystems were important enough to be included in such a high-level architecture and in how these subsystems interact with each other. However, after discussion, we determined that the most critical subsystems are the User Interface, Memory, Network, Plugins, Browser Engine, Render, and JavaScript V8.

When determining the interactions between the subsystems, we initially thought that the Browser Engine encapsulated all processes, and therefore we had all of the subsystems created with dependencies to the Browser Engine. After further research and understanding of how the multi-process architecture works, we realized that this was not the case and that some subsystems are only accessible through the Render. This led us to derive the conceptual architecture that is diagrammed in Figure 1.

Conceptual Architecture

Through our derivation process above, we incorporated Chrome's multi-process architecture and core principles into our conceptual architecture shown in Figure 1. The internet is filled with poorly written and potentially harmful websites that can put your browser at risk of crashing or cause a security threat to your system when rendering the content. Running multiple processes can mitigate this risk by isolating the main browser in its own process and have multiple separate processes render content from the web. This way, a

single instance of a rendering process can crash without taking down the entire browser, thus improving stability. As well, by isolating the rendering process and running it in a sandbox away from system resources, the user's system will be protected from malicious applications on the web. A positive impact on performance can also be noticed on multi-core systems since each tab runs independently.

After a long discussion and research about our component interactions, we agreed that the best representation of our conceptual architecture is a layered architecture style. The top layer is the User Interface (UI) layer which handles user input and displays content returned from the browser. The second layer consists of the Browser Engine, Memory, Network, and Plugins. The Browser Engine creates and manages render processes as well as coordinates their memory and network access. The third layer consists of the Render which is responsible for applying all logic associated with displaying a web page's resources, including handling HTML, CSS, Images, and other file types. All JavaScript encountered during rendering is handled by the final layer which consists of the JavaScript V8 component.

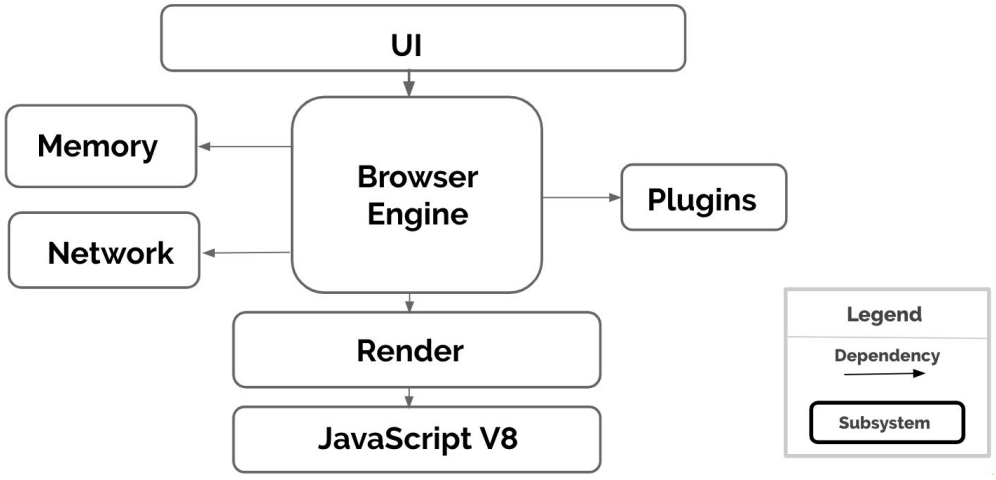

Figure 1. The proposed conceptual architecture of the Google Chrome browser.

Subsystems

User Interface (UI)

The UI is the subsystem responsible for the interaction between the user and the Chrome browser. Examples of UI components include: the address bar, forward/backward button, bookmarks, and refresh button. The UI communicates with the browser to handle user input and output to display and fetch new and dynamic content. The Chrome team wanted to create a unique UI which compelled them to develop a custom framework called views. The non-content area of the UI is made up of a tree of views that handle rendering, layout, and event handling. The content area of the UI is implemented with HTML technologies and should require little to no internet connection.

Browser Engine

The Browser Engine is a layered subsystem which is the central coordinator between the conceptual architecture's subsystems. The main intention of the Browser Engine is to control the actions between the UI and Render.

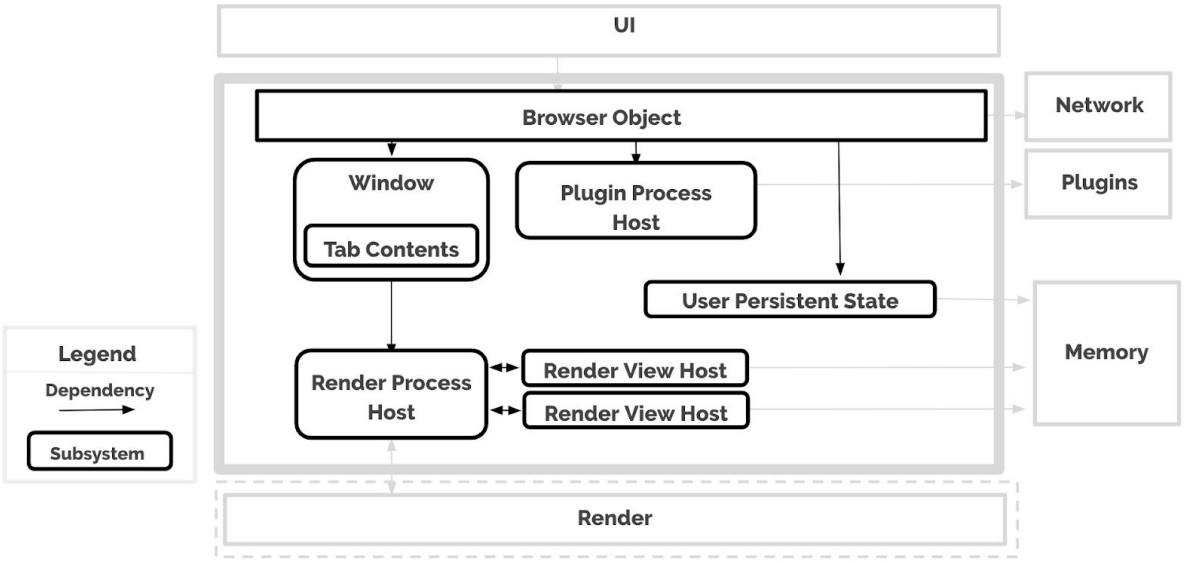

Figure 2. The proposed subsystem of the browser engine.

Browser Object & Window

The Browser Object represents a top level view of the Browser Engine. It acts as a client to the other components within the Browser Engine and passes information between them. The Browser Object connects to the Network to fetch the requested resources given by the URL from the UI. The Browser Object relies on the Window, which manages all the tabs that are open in the browser. Window initiates the rendering process for a tab by calling Render Process Host with the given resources returned from Network.

Render Process Host & Render View Host

The Render Process Host controls the rendering process (displaying) of a web page. When a new tab is opened, it creates a new, individual Render View Host. The Render Process Host communicates to render through IPC and ensures Render runs in isolation and only gains access to permitted resources. Each Render View Host corresponds to an individual Render View within Render and the Render Process Host manages the data flow between them. To ensure the speed of the browser, the Render View Hosts connect to the memory for the *backing store* in the cache. The backing store is a bitmap of the last rendered version of the page and ensures that the browser won't hang and the user will always see meaningful content, even if that means the user is not seeing the most up to date version of the page. The Render View Host allows unrelated tabs to run their processes in parallel; this

allows them to fail independently and not crash the entire browser if one process stops working.

Plugin Process Host & User Persistent State

The Plugin Process Host connects to the Plugin subsystem and manages individual processes of the plugins running in the browser. The goal of running plugins in a separate process is to protect the security of the user's system.

The User Persistent State refers to the managing of data which remains in the browser after the user has closed the page. It manages cookies, passwords and bookmarks. This data is stored using the Memory subsystem which stores data on the user's hard drive.

Memory

The Memory component is used to manage and save data for a user on the user's disk. This includes saving bookmarks, history, cookies, cache of frequently/recently accessed pages, and all other data required to persist after the user has closed the browser. It is important to note that the memory component talks directly to the Browser Engine and not Render. This way, system resources can be isolated from the Render which can potentially do harm to the user's file system. As well, saved data can be shared and accessed by each given render process via the Browser Engine, this allows the shared data to be consistent among each tab in the browser.

Network

The Network component is used to connect to remote servers and fulfill URL requests given by the Browser Engine. Most URL requests are HTTP (http:// or https://) requests which will return the resources of a webpage (often HTML, CSS, and JavaScript files) from a remote web server to the Browser Engine which in turn delegates rendering to the Render process. The Network component handles a variety of other less common requests such as file:// requests, ftb:// requests, and data:// requests. Network also stores resources fetched on the web into cache in the Memory component via the Browser Engine. This allows recently requested resources to be quickly accessed at a later time.

Plugin

Plugins provide additional features to the web page that the browser cannot handle itself, such as Adobe Flash and Java applets. They exist in their own subsystem for security and maintaining compatibility. Since a plugin may require privileges like file access, it is not contained in the sandboxed render. As well, if there is a crash caused by a third-party plugin, the web page is not compromised. Chrome is designed to handle three types of plugins: out-of-process, in-process, and windowless. For each unique plugin, there is one plugin process and many render processes through the browser engine. By keeping these plugins in their own process the Chrome browser is more stable.

Render

The Render is responsible for controlling the contents of the tab or window displaying a website. The Render is specific to each tab or window, so if one tab crashes the other tabs will remain functional. If the number of processes is too large, a single Render will be used for more than one tab. This subsystem has an object-oriented style, allowing changes to be made within certain objects without affecting the entire system.

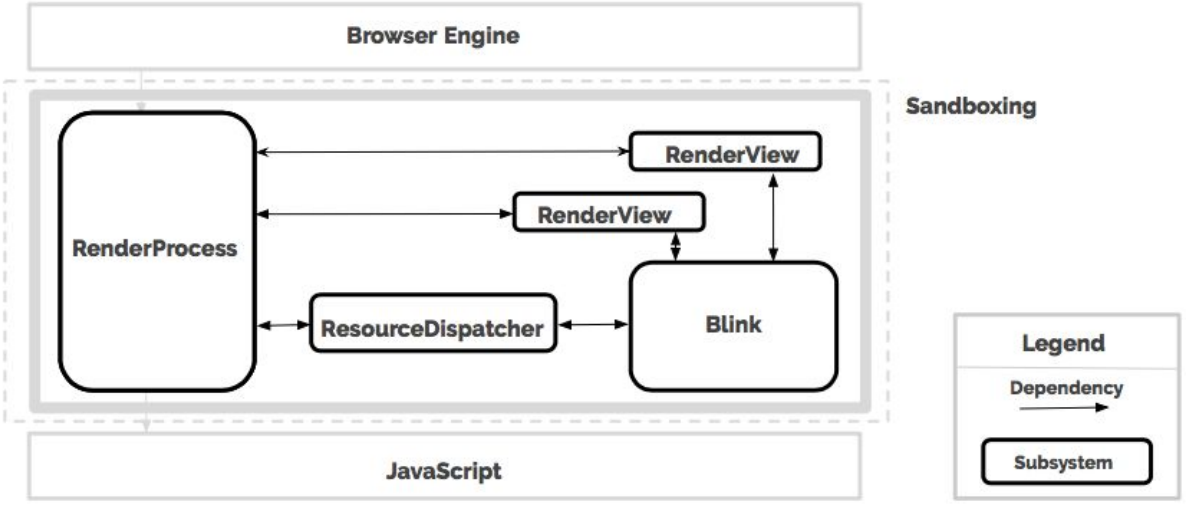

Figure 3. The proposed subsystem of the render within the conceptual architecture.

RenderProcess & RenderView

The Render consists of a *RenderProcess* that communicates with the browser. There is one *RenderProcess* per render process, so the browser will call to it when a new tab or window is opened. It is connected to one or more *RenderView*. The *RenderView* corresponds to the content of the tabs. It is managed by the *RenderProcess* and gathers web content from Blink.

Blink

Blink is responsible for implementing the content inside of the tabs. It replaced *WebKit* in 2013 when it was developed by The Chromium Project. Blink implements the specs of the web pages, builds the document object model (DOM) trees, calculates style and layout and requests resources from the underlying stack network.

ResourceDispatcher

The *ResourceDispatcher* exists due to the *RenderProcess* not having access to the internet. It is employed when a server is needed to fetch content for a webpage. The request is sent to the browser via the *RenderProcess*. It is also used to connect to the browser and

dispatch system calls when Blink requests file access or audio playing, as well as when Blink requires user profile data like cookies, passwords, etc.

Sandboxing of Render

The Render subsystem is sandboxed which allows it to run in a separate process and restrict access to system resources. Since the Render interacts with potentially untrusted web content more directly, it's security is important. It only has access to the user's file system via its parent browser's process. The access to the user display and related objects can be limited. Thus, a process could be run in a separate window that is not visible to the user. As well, sandboxing the Render helps protect the security of the browser against hackers. System calls that could assist a hacker are restricted from the Render.

JavaScript V8

V8 was developed by The Chromium Project to interpret JavaScript. The JavaScript code embedded in the website to be displayed will be interpreted and executed by V8. The interpretation by V8 is shared with the Render to be displayed. It is dynamic as it can run standalone, allowing additional features to be added to JavaScript. Since JavaScript has no classes and is dynamically typed, hidden classes are created at runtime. This creates an internal representation of the type system and can improve the property access time. It is independent of the browser and exists per tab or window. If one tab were to freeze the other tabs would still be functional to execute the JavaScript.

Use Cases

Use Case 1: User successfully logs into a website, Chrome saves the password

The sequence diagrams for the use cases were built using the main components of the conceptual architecture. For use case one, the process begins when the user opens a browser; the Browser Engine must connect to an external web server via the Network component to load the initial page's resources. From there, the Browser Engine renders the webpage through the Render component. Once this is completed, the process returns to the UI so that the User can now log in to the webpage. When the user logs in to the webpage, the process goes from the Browser Engine to the Render component. From there, it goes to the JavaScript V8 component where the user credentials are validated using the webpage data. Next, the process will return to the UI component to inform the user that their login was successful. Finally, the process will continue to the Browser Engine where it will then save the password in the Memory component.

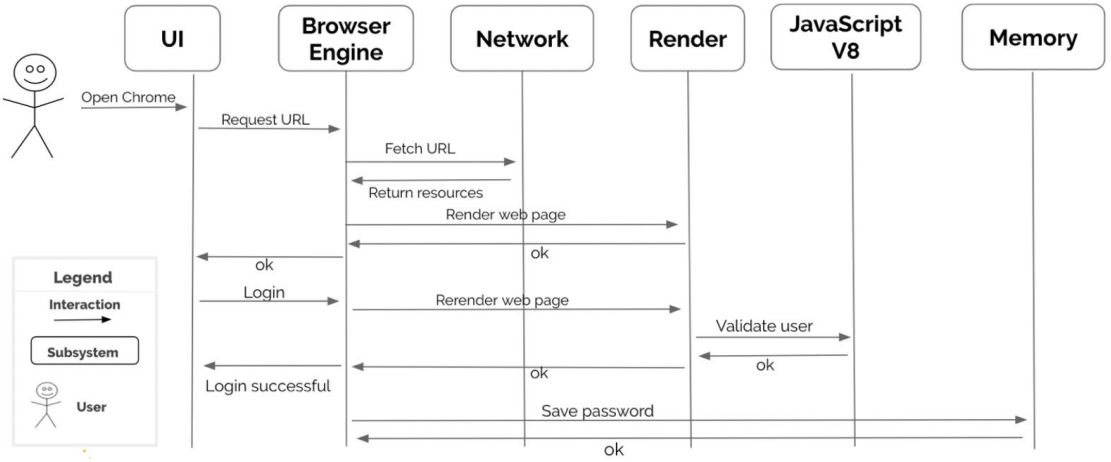

Figure 4. Sequence diagram for a user logging in then Chrome saving the password.

Use Case 2: Chrome renders a web page that has JavaScript

The second use case concerns a user requesting a web page which has JavaScript. The user opens a web page using the UI and types a URL into the address bar (or clicks a link). The UI sends the requested URL to the browser for display. The Browser Engine makes an HTTP request to the Network for the resources. The Network returns the page and its resources to the Browser Engine. The Browser Engine then calls the Render to display these files according to their content. The Render uses the JavaScript V8 engine to interpret the JavaScript files. The Render then displays the web page on the screen and returns an ok signal to the browser.

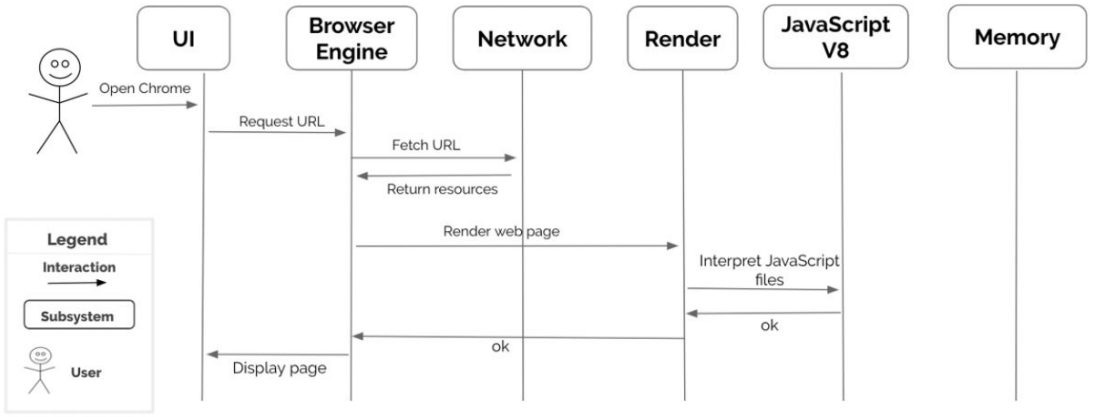

Figure 5. Sequence diagram for Chrome rendering a web page that uses JavaScript

Concurrency Model

During development, the Chrome architects worked to develop a concurrency model that would support their system. A concurrency model determines how threads in a system collaborate and communicate to get the job done.

For collaboration, Chrome implemented a multi-process architecture. For each tab that is opened or plugin that is run, the Browser Engine starts a new process. From there, the Render of each process, which operates inside a sandbox, renders all content to the new tab. As an added layer, Chrome supports multi-threading within each individual process. Each process has its own main thread and IO thread and can handle more threads for various other specialized functions depending on the process requirements. The main thread in the Browser Engine updates the User Interface which the user is interacting with on their device. The main thread in Render runs the entire rendering engine. Currently, Chrome uses a software called Blink to optimize the main thread and run the important processes involved with the Render process. The IO thread in the Browser Engine handles the Inter-Process Communications and any network requests. In Render, the IO thread handles only the Inter-Process Communications.

For communication in Chrome's concurrency model, the architects use Inter-Process Communications or IPCs. These protocols, formerly the Chromium IPC (still being used at lower levels of the architecture), have now been converted to the Mojo system. Mojo allows cross-process and cross-thread communication by message passing using message pipes. The individual processes do not share memory meaning they cannot communicate using shared variables and instead must communicate by sending and receiving messages. Message passing is used for invoking behavior and involves using an object model to determine the function from the implementations. It relies on the object to select and execute the appropriate code once the invoking process has sent a message. This system allows messages to be passed and commands to be executed either across process boundaries or between threads in the system.

Ultimately, using a multi-process architecture allows for higher performance with increased stability and security for the device. This does not come without its disadvantages. By creating multiple processes, there is an additional fixed cost of RAM usage for each process. This is a result of Chrome duplicating some tasks and plugins in each tab, which uses more memory, but reduces memory bloat and allows appropriate memory allocation over the long term.

Team Issues Within Chrome

Chrome is developed with several teams working concurrently to create a seamless web browsing system. As the Chrome project grew larger throughout development, it became more challenging to maintain a strictly layered system as subsystems were being updated and reconfigured regularly within each team. This also created various dependencies, which are connections in the system, where there should not have been any. Because of these challenges, the Chrome team was forced to change the way they stored their code so that each team could work more independently. Also, the Chrome architecture had to be updated on

many occasions as technology improved. It is important for the Chrome Browser to have the same functionality across all of its user interfaces and platforms which is a large issue that the Chrome team has been grappling with from the day they started developing.

Current Limitations and Lessons Learned

Although Chrome is 10 years old, there is still very little documentation regarding the high-level architecture available on the internet. Our team found it challenging to find updated information on the Chrome browser and the associated open-source Chromium Project. Generally, the information was either quite technical with source code or very basic and unhelpful which made it challenging to determine the high-level conceptual schema of the architecture. Furthermore, the information was often outdated with older subsystems or technologies being used to explain large concepts of the Chrome architecture.

As a team, we learned many things about ourselves as individuals and as a team while working through this project. We learned the importance of communication throughout every aspect of the project. We found we did our best work together collaboratively with lots of discussions as opposed to working independently and coming together for the final product. We also determined setting goals and deadlines for each aspect of the project was valuable so that we had enough time to provide feedback and create a final product that satisfied each member. For our next project, we will spend more time researching and learning about the topic before we begin working on the final product.

Conclusion

After a significant amount of research and collaboration, our group worked through the various limitations we faced to conclude that Chrome's conceptual architecture is layered at a high level. Furthermore, within the Browser Engine we found the structure to also be layered, while the Rendering Engine uses an object-oriented style. This mainly layered architecture, along with the well designed concurrency model and multi-process architecture, has allowed Chrome to succeed in its goal of being a simple, fast, stable, and secure web browser that results in a positive user experience.

In its ten years of being on the market, Chrome has advanced significantly as a result of it being continually refined and improved everyday by a dedicated team. Our group is intrigued to see if in 10 years time, Chrome will continue to dominate the web browser world, or if another company will have the capability of creating a new browser or enhancing a current browser that will be able to outperform Chrome.

Data Dictionary

Architecture: High level structure and organization of subsystems in a software.

Browser: Program used to display web content and allow for user interaction.

Cache: Allows faster access of recently/frequently accessed data by storing it in a smaller, faster memory.

Cookie: Data created by a website about a user to be temporarily or permanently stored in the user's device as a text file for personalization of web content.

JavaScript: Programming language used to create dynamic web pages.

Layered: Architecture style that organizes subsystems hierarchically.

Open-Source: Source code of the program is freely available to the public to be viewed and modified.

Subsystem: Component of a larger system.

Threads: Units of a program which are running simultaneously.

Naming Conventions

Document Object Model (DOM): Represents the web page as nodes and objects to modify the structure, style and content.

HyperText Markup Language (HTML): The markup language used to display contents of a web page. It controls the layout of the page.

HyperText Transfer Protocol (HTTP): The protocol used by the World Wide Web defining how messages are formatted and transmitted between the browser and a web server.

Interprocess Communication (IPC): A mechanism that allows processes and threads to communicate between one another. This can be unilateral or bilateral depending on the system.

Random-Access Memory (RAM): A type of computer data storage that stores data and machine code currently being used. Allows for quick read and write access to the storage device.

User Interface (UI): The subsystem in a software where the user interacts with the software.

References

S. (n.d.). Browser Market Share Worldwide. Retrieved October 12, 2018, from http://gs.statcounter.com/browser-market-share

Grosskurth and Godfrey. (n.d.). A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser. Retrieved October 13, 2018, from http://research.cs.queensu.ca/~emads/teaching/readings/emse-browserRefArch.pdf

G. (n.d.). Design Documents. Retrieved October 12, from http://www.chromium.org/developers/design-documents

G. (n.d.). Google Chrome Comic Book. Retrieved October 11, from https://www.google.com/googlebooks/chrome/index.html

C. (n.d.). Threading and Tasks in Chrome. Retrieved October 13, from https://chromium.googlesource.com/chromium/src/+/lkgr/docs/threading_and_tasks.md

M. (n.d.). Mojo System. Retrieved October 13, from https://chromium.googlesource.com/chromium/src/+/master/mojo/README.md

Fisher, D. (2009, April 29). Retrieved October 19, 2018, from http://www.youtube.com/watch?v=A0Z0ybTCHKs

Plugin Architecture. (n.d.). Retrieved October 19, 2018, from https://www.chromium.org/developers/design-documents/plugin-architecture

Multi-process Architecture. (n.d.). Retrieved October 19, 2018, from https://www.chromium.org/developers/design-documents/multi-process-architecture

Core Principles. (n.d.). Retrieved October 19, 2018, from https://www.chromium.org/developers/core-principles