CISC 322

Assignment 1
Conceptual Architecture of Google Chrome

The Koalas
Aawista Chaudhry 18ac20 20129987
Alexander Salgo 18ajbs 20138440
Emily Kaczmarek 13ek38 10138915
Matthew Eliot 12mne1 10100789
Rita Alsattah 14ra6 10154610
Scarlett Taviss 12st65 10092062

ABSTRACT

This report gives an overview of the conceptual architecture for the Google Chrome web browser. Chrome is a product of the open-sourced Chromium Project. Documentation by the Chrome Development Team, and other sources, was examined to derive the proposed conceptual architecture. It was determined that Chrome has an object-oriented architecture style comprised of five components: User Interface, Browser Engine, Rendering Engine, Plugins, and Networking. Each of these subsystems has their own unique architecture and can be split into many subsystems. The pipe-and-filter architecture of the Rendering subsystem is examined in this report.

Chrome implements multi-process architecture, a significant difference from the web browsers preceding it. This allows for many Renderers to be run in parallel, normally one for each website the browser is accessing. By increasing concurrency, this gives Chrome many positive attributes including increased speed and stability.

Inter-process Communication (IPC) was developed to enable the components within Chrome to communicate, acting as a series of pipes connecting the architecture. When the browser performs a task, IPC allows for easy tracing of requests and returns between the subsystems.

Chrome is a very complex system that is more commonly analyzed at a lower level than architecture. As a result, experts have differing opinions regarding its architecture. The derivation process for this report was limited by the information available and the complexity of the system. Furthermore, the proposed architecture reveals many issues that the original Chrome Development Team would have encountered.

Table Of Contents

1. Introduction & Overview

Google Chrome is a revolutionary browser. Over the ten years of its existence, Chrome has introduced many concepts to the browser landscape that were previously unheard of. These range from its search bar integration (Chrome Team, 2010), to its championing of HTML5 (Crawford, 2011), and even its recent visual redesign. Arguably its most crucial innovation, however, is the unique architecture that underlies its software. This report examines the conceptual architecture that would have acted as a guide for developers during the software design process.

As the name implies, the conceptual architecture being investigated is not a concrete aspect of Chrome. While developing this report, no source code was examined. Instead, sources were consulted that discussed flow of data within Chrome at a high level of abstraction. Due to this abstraction, opinions among experts differ regarding the organization of Chrome's software. Being a program that is ten years old (with source code containing over 24 million lines of C++, HTML (HyperText Markup Language), JavaScript, and other languages), the flow of information within Chrome is never perfectly captured by single-page diagrams (OpenHub.net, 2018). Nevertheless, this report provides a useful and meaningful representation of how the code behind Chrome is intended to function.

In addition to proposing a software architecture for Chrome, this report will explain the subsystems involved. Each subsystem has a variety of related functions, and connects to other subsystems in a well-defined manner. The functions and connections present will be explained and justified. The subsystems involved include: a User Interface (UI), a Browser Engine, a Networking component, multiple parallel Renderers, and a Plugins module. These subsystems are arranged into an object-oriented architecture; the reasoning behind this will be discussed in the architecture section.

The architecture section will also provide an overview of the derivation process by which the final architecture was arrived at. This includes the research, collaboration, and prototyping of different architectures. Three examples of architectures that were considered during this derivation process will be provided, alongside an explanation of how they differ from each-other and from the final architecture.

Just as Chrome has an architecture, each subsystem within Chrome is complex enough to have its own architectural representation. This report examines the pipe-and-filter architecture that comprises the renderer subsystem, and explains how this subsystem interacts with the rest of Chrome. Unlike the other subsystems, there can be multiple rendering subsystems running concurrently at any given time within chrome.

Chrome implements this massive concurrency by using a multi-process architecture. This architecture takes advantage of an inter-process communication (IPC) system to allow for coordination of multiple renderer subsystems, each with its own associated process. This gives Chrome a significant advantage over other browsers in the form of improved stability. In the concurrency section of this paper, this aspect of Chrome is discussed in-depth.

Two use cases for the architecture analysis were given and made into sequence diagrams. These diagrams illustrate the flow of information between subsystems during different program functions. Sequence diagrams are included for two use cases: a user logging into a website and saving their password, and Chrome rendering a web page that includes JavaScript.

Designing a conceptual architecture also facilitates speculation regarding the issues that the Chrome Development Team would have faced during software development. In Section 7, this report examines two potential collaboration issues that may have arisen during development due to the software architecture. The first issue involves coordination between developers working on the Renderer and those working on the main Browser Engine. Issues would have occurred here during the development of the multi-process architecture discussed above. The second issue examined is the coordination of both the Renderer and Browser teams with developers working with plugins. This would have been required due to the Plugins subsystem communicating similar information to both other subsystems.

The report will conclude by discussing the lessons that can be learned by performing an analysis of the type performed here. This includes things that can be learned about the specific conceptual architecture being analyzed, as well as what can be learned about architecture analysis in general. The limitations faced by this study will also be discussed here, including the limitations of the final conceptual architecture.

2. Conceptual Architecture

2.1. Final Conceptual Architecture

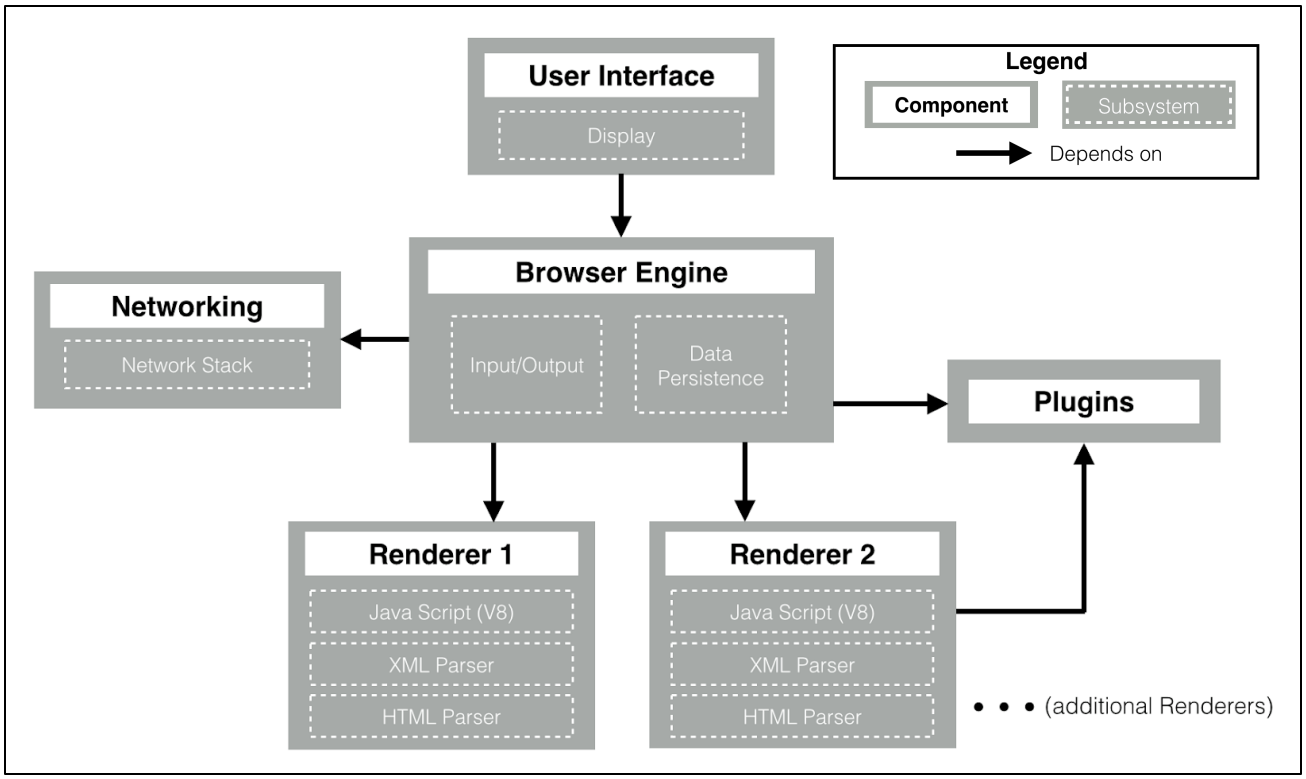

Figure 1: Final conceptual architecture of Google Chrome derived using the process described in Section 2.2.

Figure 1 is the final conceptual architecture for the Google Chrome Browser. It is represented as an object-oriented architecture style, as this was deemed the best fit for the subsystem dependencies. A lot of the Chrome Browser is about protecting the security of the user device, and object-oriented style is all about protecting related information and encapsulating data representations and operations (Hassan, 2018). The Browser Engine is the central subsystem that communicates with all others. This architectural style is advantageous because the implementation of subsystem can be adjusted and updated without affecting the entire system as a whole (Hassan, 2018). Each subsystem and their dependencies are discussed in detail late on in this report.

2.2. Derivation Process

The architecture proposed was arrived as via a 3-step derivation process. The three steps involved were research, prototyping, and collaboration.

During the research step, sources were consulted to obtain information about the conceptual architecture behind Chrome. In order to reduce the likelihood of obtaining incorrect information, the sources were limited to five types:

1. *Google Development Team Articles* - Articles written by Google employees regarding either the design and development process or the architecture itself.
2. *Expert Analyst Articles* - Articles written about Chrome's architecture by software experts.
3. *Chrome Developer Documentation* - Documentation about Chrome provided by Google for current app developers.
4. *Google Conference Videos* - Recordings of presentations by industry experts and Google employees regarding Chrome's architecture.
5. *Web Browser Reference Architectures* - Standard reference architectures for web browsers.

After consulting these sources, team members began the prototyping step; they were split into three groups of two members. Each group constructed their own architecture for Chrome, which they believed was best supported by the research they had conducted. These alternative architectures are discussed in-depth in the section below.

The final step, collaboration, involved all teams comparing and contrasting their models. The models were compared based on how well they explained the behaviour of Chrome during different use cases, as well as how supported they were by sources. Although sources sometimes supported different architecture models, this step allowed for a final architecture to be constructed which best fit all use cases and sources.

2.3. Alternate Architectures

All three alternative conceptual architectures shared certain common features. Each had a central Browser Engine that coordinated the interactions between different subsystems. The Browser in each architecture was dependent on multiple Renderers, highlighting the multi-process

architecture of Chrome. Each architecture also had a Plugins subsystem; however, the dependencies on this subsystem varied between the three designs.

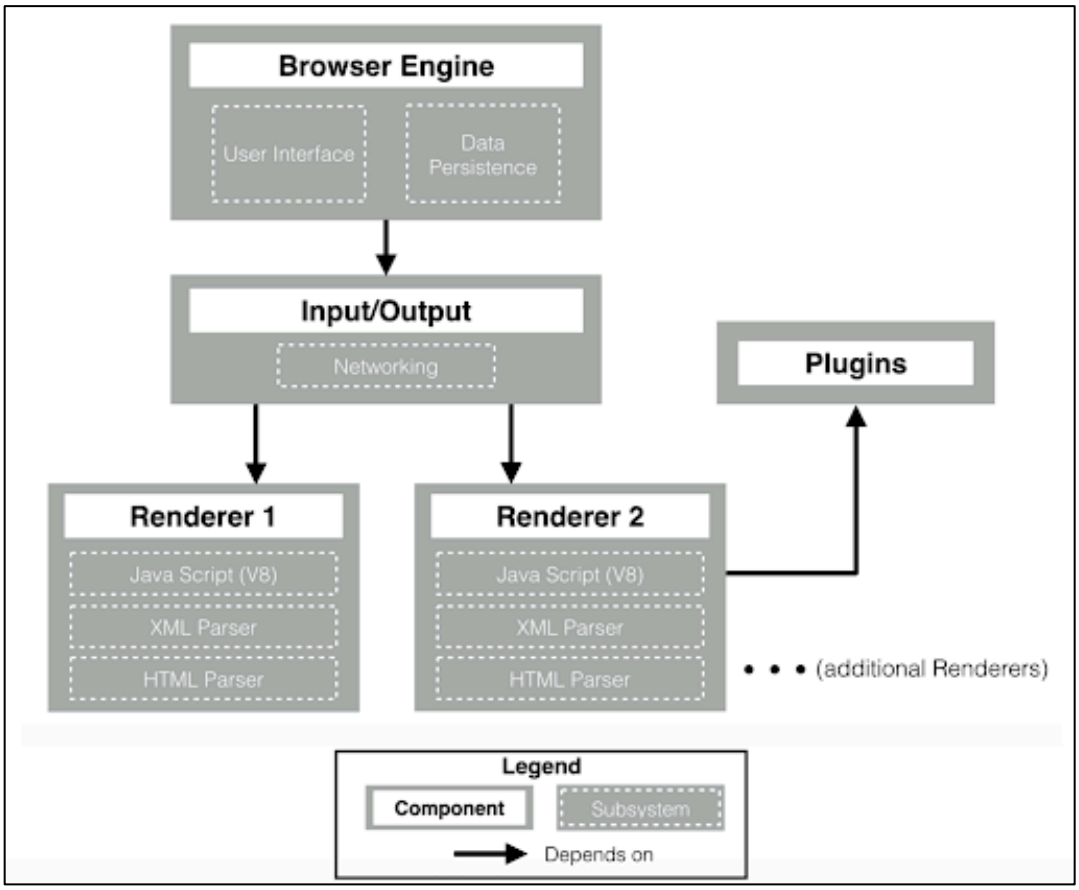

Figure 2: Alternative conceptual architecture 1, designed as a layered architecture style.

The first alternative conceptual architecture created is shown in Figure 2. The subsystems here follow a strictly layered architecture, meaning that each layer calls procedures from layers below it, and receives procedure calls by the layer above it (Hassan, 2018). The layers are therefore displayed in a hierarchy, where only certain elements of the higher levels are visible to the lower levels. The advantage of this architecture is the reusability, since layers can be implemented in different ways without affecting the other layers. This architecture style was not chosen for the final design, due to the function of the input/output subsystem. It was originally believed that this subsystem would solely handle networking, and could therefore act as a link between the browser engine and the renders. The browser would rely on the input/output to collect the information required to load a new webpage, which would then rely on the renderer to create the design of the webpage. After consulting further documentation, it was discovered that Input/Output (I/O) handles both networking and inter-process communication between the Browser and the Renderers (The Chromium Project, Inter-process Communication (IPC), 2018). For this reason, Input/Output was placed as a subsystem within the Browser, which communicates with both the Renderers, and a separate Networking subsystem. Due to the

Browser now being dependent on both the Renderers and Networking, the architecture can no longer be layered, since the system cannot be represented in a hierarchy.

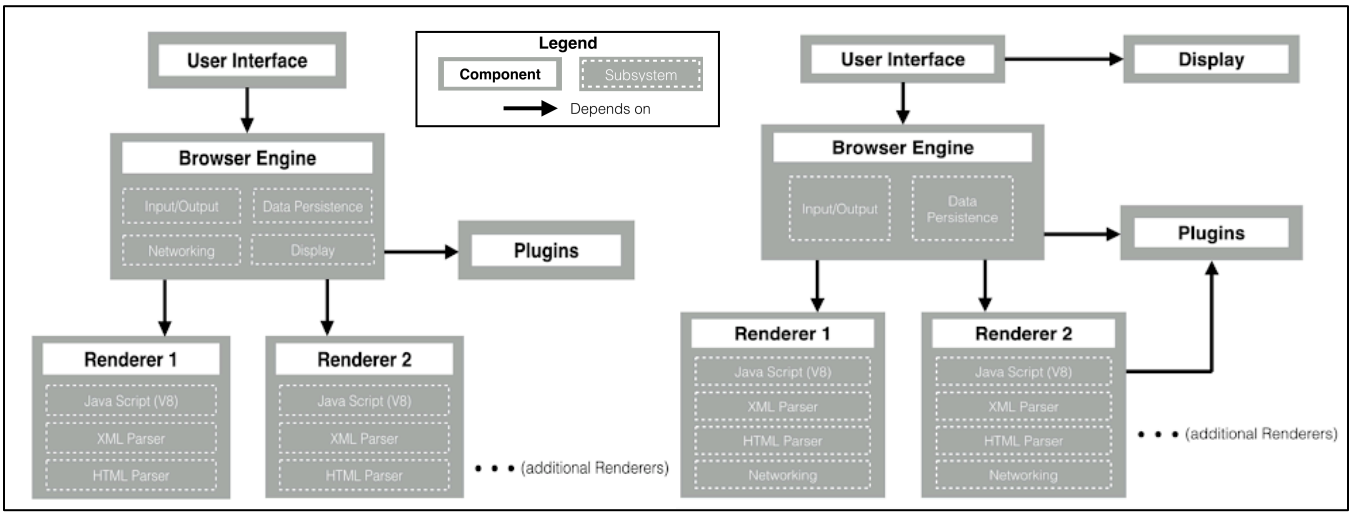

Figure 3: Alternative conceptual architectures 2 & 3, designed as object-oriented architecture style.

The second and third conceptual architectures created were both object-oriented, as shown in Figure 3. Each subsystem is an object that has related subsystems within it (Hassan, 2018). These objects are connected to each other with function and procedure calls. The object-oriented architecture style also allows different implementation of subsystems to be created without affecting others, but does not require subsystems to be organized in a hierarchy, as the layered architecture style does. It was decided that object-oriented architecture therefore better suited the architecture of Chrome, and it was chosen as the style for the final conceptual architecture. The differences between the two object-oriented conceptual architectures were then discussed to determine the final conceptual architecture design. It was decided that Display should be a subsystem within the User Interface, since the User Interface is where calls to the operating system are made to present data on the screen. The dependencies on the Plugins subsystem were also discussed. The final decision was to have both the Browser Engine and Renderers dependent on the Plugins. This was due to the wide variety of plugins available for Chrome. Certain plugins affect the Browser Engine directly, while other plugins will affect the rendering of the layout on the screen (Reis, 2008).

3. Architecture Subsystems

3.1. User Interface

The main component of the UI is the tab, which has been flipped upside down from the normal convention for web browsers. Each tab can easily be moved to and from different windows, has their own control buttons, and its own Uniform Resource Locator (URL) box. This is in contrast to most other browsers, who have one toolbar and URL box with multiple tabs below them. The goal for the user interface was to be as invisible as possible to the user so no interruption or distraction from it would occur. Web applications can be launched in their own streamlined

window, with the toolbar and URL box removed for a better user experience with the browser. (Chrome Team)

The User Interface is where the user will interact with the Browser Engine; all communication with the browser takes place on the UI. Browser operations such as reload, back button, forward button, etc. and webpage requests begin on the user interface. When a user interacts with the UI, a message is sent to the Browser Engine to handle the event. Once the requested web page, or button click action, etc. is finished, the browser returns the displayed page to the UI. The User Interface is only dependent on the browser to retrieve all information for displaying the requested event. (Garseil & Irish, 2011)

Display of the User Interface is handled within this subsystem. The way it is shown on an individual's computer depends on the operating system and user preferences. Users can modify the way a browser looks, hence, modifying the user interface, which will be stored within the UI's subsystem called Display.

3.2. Browser

The Browser is the most central component of chrome because it interacts with all other subsystems in some way. As mentioned previously, the UI depends on the browser to relay the requested event to the appropriate subsystems to complete it. The Browser is dependent on all other subsystems in the architecture, which includes the Networking subsystem, the Rendering Engine, and Plugins. It depends on the Networking subsystem to find the requested URL on a web server and relay it the code makeup of that web page; the Browser can then relay this information to the Rendering Engine for processing. The Rendering Engine is something the Browser is heavily dependent on because it compiles all the information for how the requested URL is displayed and relays that back to the browser. Lastly, the browser is dependent on Plugins for auto updates and so that frequently used and cached websites requiring plugins won't need to go through the rendering process to be displayed on the User Interface.

The Browser has two subsystems of its own, I/O and Data Persistence. The I/O subsystem handles all the communication between the Browser and the UI, the Renderer, the Plugins, and the Networking subsystems respectively. Data Persistence stores data within the Browser Engine; this could either be low-level data such as cookies and caches or high-level data such as bookmarks, and toolbar settings. (Grosskurth & Godfrey, 2007)

3.3. Networking

The Networking subsystem handles all aspects of communication and security with the Internet. It deals with all URL requests made by the Browser and is responsible for retrieving its information from a web server. URLs are requested using file transfer protocols like HyperText Transfer Protocols (HTTP) and File Transfer Protocols (FTP). (Grosskurth & Godfrey, 2007)

Although almost, if not all, the information returned to the Browser from the Networking subsystem is sent to the Renderer, it cannot directly depend on Networking. Not having direct access prevents malware and phishing attacks from occurring on your computer through the web

and any bugs that could be occurring on either side won't be able to be transferred in during the process. (Chrome Team, 2008)

3.4. Renderer

The Rendering Engine interprets the HTML, eXtensible Markup Language (XML), and JavaScript that comprises a URL and generates the layout that is displayed on the UI. Found within the Rendering Engine is the HTML parser, XML parser, and V8 JavaScript interpreter to handle all tasks. The Renderer holds the global state of the system as it is the only subsystem, apart from third party plugins, that can read the code for displaying things on the UI. All rendering processes are sandboxed to ensure malware isn't installed on one's computer and can't affect more than one tab. Each render process can compute, but cannot read or write files to a user's hard drive. (Chrome Team, 2008)

Renderers are only dependent on Plugins, which is case dependent. If a website requires a plugin, for example Adobe Flash, it is sourced out to retrieve the necessary information for the Rendering Engine to complete the layout of its display. Plugins are not found within the Rendering Engine, as they cannot be fully sandboxed (Chrome Team, 2008).

3.5. Plugins

Plugins are primarily used to enable third party developers to create abilities that extend those of the browser; some examples of plugins include Adobe Reader, Adobe Flash Player, and QuickTime plugin. A new process is created for each plugin that is in use, each having an interface for communication with the Browser and Renderer. Once a user is finished with the tab(s), the process is destroyed. By keeping Plugins in a separate process than the Renderer, the rest of the tab can be sandboxed, even if the plugin can't. (Chrome Team, 2008)

4. Renderer Subsystem Internals

The Renderer Subsystem has a pipe-and-filter architecture, as shown in Figure 4, and it is responsible for creating the visual representation of a given URL by parsing the XML, HTML and Cascading Style Sheets (CSS) code, construction of a Document Object Model (DOM) and V8 Instance (JavaScript execution) for a tab. It processes the web page to bitmaps and sends it back to the Browser, which displays the final construction of the web page on the UI (The Chromium Project, The Rendering Critical Path, 2018). The Renderer subsystem also maintains a global state for each tab, allowing for information on a page to be changed and for events such as mouse clicks to be handled properly. Each of the Renderer subsystems is sandboxed in order to have restricted privileges. The Renderer subsystem is only dependent on Plugins, when applicable, as they cannot be sandboxed within the engine itself and must be called by the renderer.

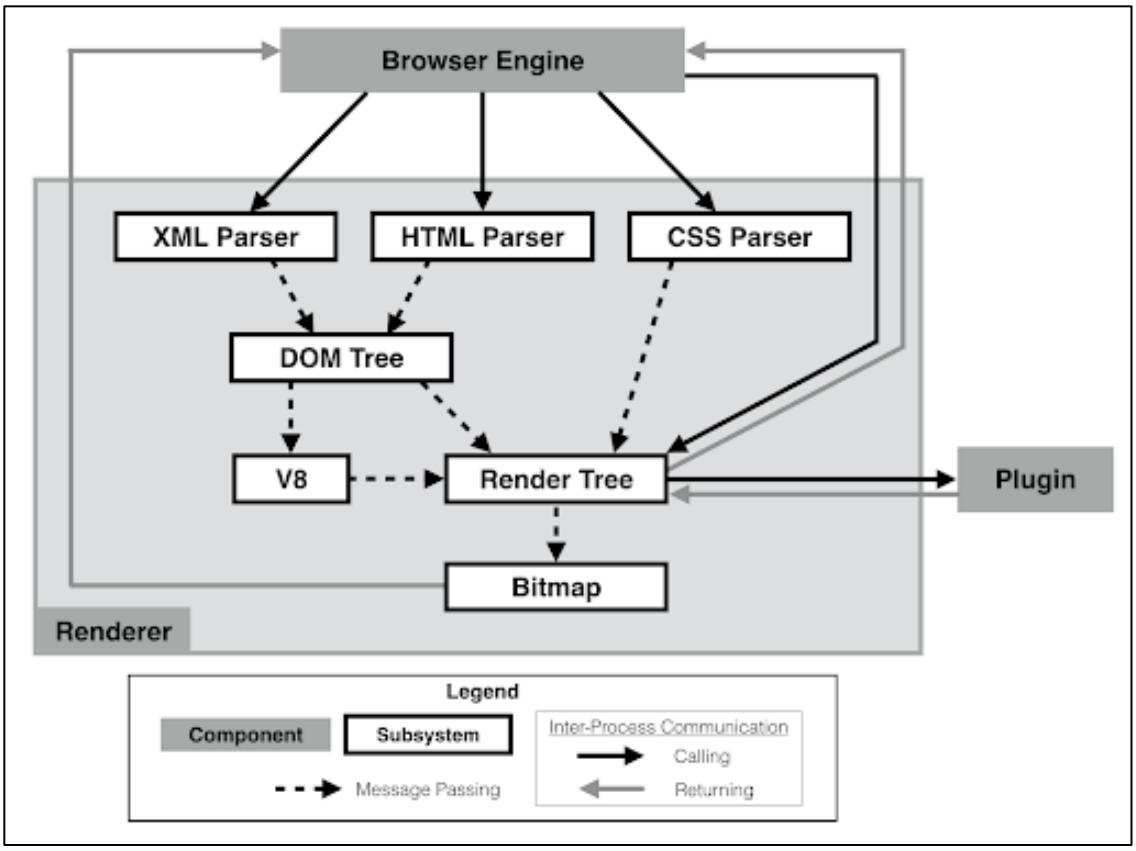

Figure 4: A closer look at the pipe-and-filter architecture of the Renderer subsystem and its dependencies.

4.1. XML, HTML, and CSS Parsers

The Browser first makes a call to the Renderer by sending it 3 pieces of information: the XML, HTML, and CSS data. The Renderer then sends each to their own specific parser, which is designed to read the uninterpreted representation of the data in the files and convert it to a form that is usable by the rest of the program. These are DOM trees for XML/HTML and CSSOM for CSS. (Grigorik, Constructing the Object Model, 2013)

4.2. DOM Tree

The DOM trees for HTML and XML are combined into a single DOM tree here (if both have been generated). The final DOM tree captures the properties and content of the page (The Chromium Project, The Rendering Critical Path, 2018). If any JavaScript objects are present in the DOM tree, it makes a call to the V8 interpreter to parse these.

4.3. V8 JavaScript Interpreter

V8 is a powerful open-source JavaScript engine provided by Google. It was first built to enhance the performance of JavaScript execution inside web browsers by translating Java code efficiently into machine code before executing it (DroidHead, 2017). In the case of any JavaScript existent in the files, that is sent to the V8 JavaScript interpreter to be compiled before execution.

4.4. Render Tree

The CSSOM tree, DOM tree, and interpreted JavaScript are combined to construct one render tree which encapsulates the DOM content along with the CSSOM style information for each node required to render the web page (Grigorik, Constructing the Object Model, 2013). This render tree will later serve as the input to the layout process that eventually renders the pixels to the screen (Grigorik, Render-tree Construction, Layout, and Paint, 2013). This subsystem also handles the global state of each tab, allowing for information within a tab to be changed. Finally, this subsystem receives direct input from the Browser in order to allow each tab to respond to events. When a user enters text or clicks on a button on the page, the Browser makes a call to the render tree subsystem, which determines (and returns) the appropriate response.

4.5. Bitmap

The final stage of the rendering process is "painting" the web page. Following the construction of the render tree, it goes through a "layout" process, meaning each node is given the exact coordinates of its location on the screen (Grigorik, Render-tree Construction, Layout and Paint, 2013). This is done by converting each node in the render tree to actual pixels on the tree. The exact position and size of each object is computed and stored using a bitmap. Finally, the bitmap is then sent back to the Browser for it to display the final web page on the User Interface.

5. Use Cases

5.1. Use Case 1

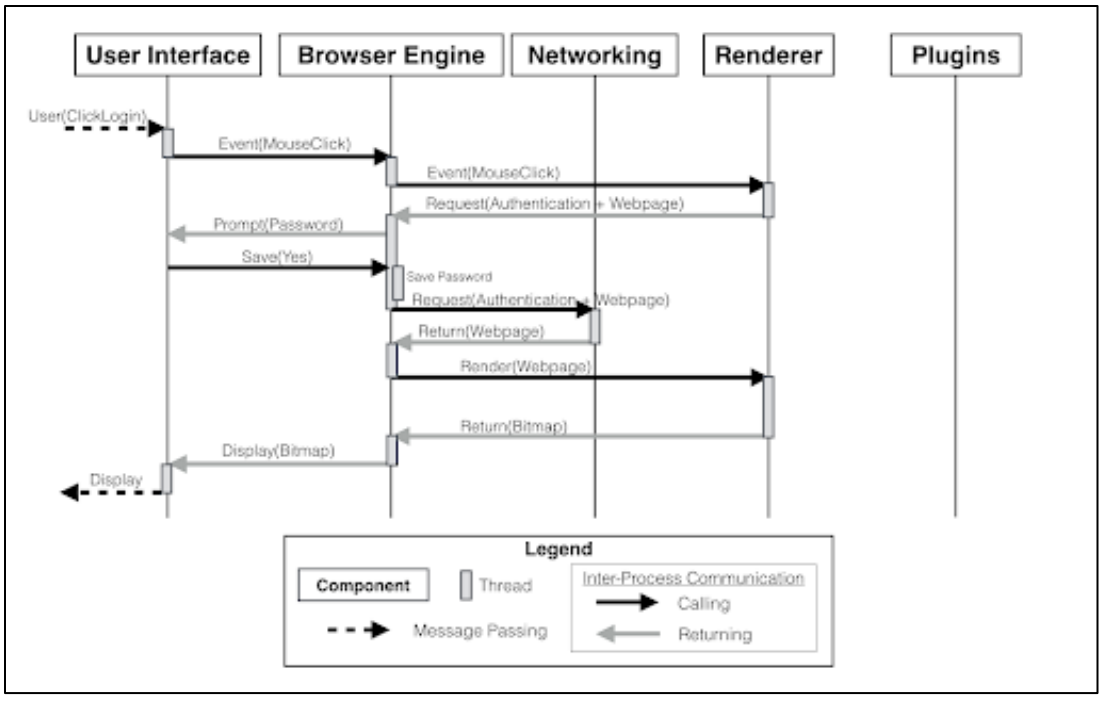

Figure 5: Sequence Diagram for a user logging into a web page successfully and having Chrome save the password.

Figure 5, which depicts the successful login and saving of a users credentials, assumes that the user has already entered their username and password into the web page. To begin the login process, the user clicks on the "login" button. The MouseClick event is then passed to the Browser Engine, which further passes it to the appropriate renderer for that page. The Renderer determines that the user clicked on the login button, and sends both an authentication request (for the login) and a web page request (for the post-login URL) to the Browser.

The Browser Engine passes this request to the Network, which communicates with the website server over the Internet via HTTP. Concurrently, the Browser Engine will prompt the user to save their password. If the user clicks on the "save" button, the Browser Engine will store an encrypted version of the username and password combination it received in its Data Persistence subsystem.

External to Chrome, the website will check to see if the information is correct and will either grant access or deny it. The website will send the result of the login request back to the Network, alongside the information for the post-login web page. The browser will pass this information along to the renderer. The Renderer uses its internal subsystems, such as parsers and a JavaScript interpreter, to render the page into a bitmap. A bitmap is a representation of the web page in a form that can be printed to the screen. Once the rendering process is complete, the bitmap is returned to the Browser, which will then return it to the User Interface to be displayed. The Plugins system is not accessed in this use case, as the webpage being loaded does not require any.

5.2. Use Case 2

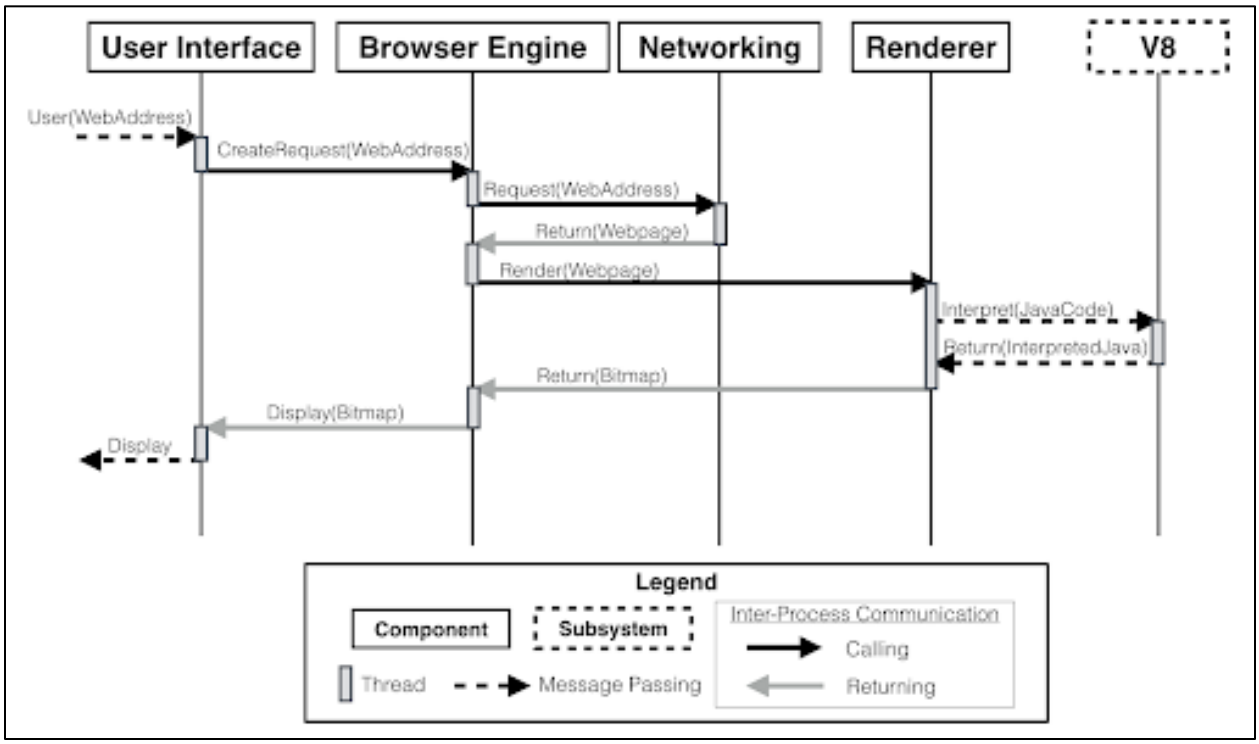

Figure 6: Sequence Diagram for Chrome rendering a requested web page that has JavaScript.

Figure 6 shows the sequence diagram for Chrome rendering a page that has JavaScript. The web page address is inputted into the URL box using HTTP. The Browser Engine receives the request, where it is then sent to the Networking subsystem to be located. Once the web address is found, the corresponding data is returned to the Browser Engine and sent to the Rendering Engine for processing. The Renderer processes the data using its subsystems as discussed previously. This specific example involves a webpage that contains JavaScript, and therefore within the rendering process the Java code is delivered to the V8 JavaScript interpreter to produce that part of the layout necessary for displaying the web page. Once completed, it is returned to the Renderer where, combined with all other components of the URL, it makes up the bitmap to be displayed on the User Interface. The bitmap is sent back through the Browser Engine and then finally outputted to the UI.

6. Concurrency

Concurrency is the ability for components within a program to be executed in parallel and without affecting each other. Essentially, a program with high concurrency can perform multiple tasks at the same time without sacrificing performance. Additionally, since tasks are running in parallel, if one fails it will not prevent the others from succeeding. Concurrency has a large impact on the speed and stability of a system, two of the most important attributes to the Chrome development team (Kurtuldu, 2018). For previous browsers, if a website in one tab crashed it could cause the entire browser to crash (Kurtuldu, 2018). Furthermore, if a website used a significant amount of resources it would slow down all other sites being run by the browser (Chintada, 2017). Chrome eliminated these problems, and many others, by increasing the concurrency of the browser (Chintada, 2017).

One of Chrome's fundamental differences from other browsers is multi-process architecture. In order for processes to run independently, the Chrome team developed Inter-Process Communication (Kosaka, 2018). IPC enables the output from one process to be the input to another. IPC can be thought of as pipes connecting the different processes, letting them communicate without relying on each other (The Chromium Project, Inter-process Communication (IPC), 2018). A significant advantage to multi-process architecture is the ability to have several rendering engines within the Browser, allowing the tabs to be run in parallel (Reis, 2008). Each rendering engine will have its own HTML parser, XML parser, and JavaScript interpreter. Usually Chrome has a separate renderer for each tab, although multiple tabs could share a renderer for a variety of reasons; this gives Chrome many positive attributes. For example, if one tab is downloading a file while another is playing a video, they will not slow each other down since they are not using the same resources and are not being executed in sequence (Reis, 2008). Furthermore, if one tab crashes it will not cause the others to crash. If the user closes a tab, instead of having to searching for the code related to that specific tab, the Browser will simply delete that Renderer (Chintada, 2017). This is a significant improvement from the previous method of running all tabs in one Renderer.

Within a process, Chrome uses multiple threads to increase concurrency (The Chromium Project, Threading and Tasks in Chrome, 2018). Threads are essentially the execution of the program within a process. Individual threads run in sequence, taking a request from a queue and executing it (The Chromium Project, Threading and Tasks in Chrome, 2018). As a result, if the thread gets

blocked it can choke the whole process. Chrome eliminated this by having multiple threads pulling requests from the same queue (The Chromium Project, Threading and Tasks in Chrome, 2018). Therefore, if one thread gets blocked it will not stop the queue from moving forward. In this way, despite each thread running in sequence, requests can still be managed in parallel. Normally, each process has one main thread, one input/output thread, a few special-purpose threads, and a pool of general use threads (The Chromium Project, Threading and Tasks in Chrome, 2018). The number of threads in the pool depends on the type of process and the resources available (Grigorik, 2013). For example, if networking has too many threads, all making requests in parallel, it could crash the router. Therefore, Chrome analyses the resource availability to determine how many threads can be used (Grigorik, High Performance Networking in Google Chrome, 2013). In general, bigger machines will have more threads and therefore will be able to make more requests in parallel, increasing concurrency.

7. Team Issues

Team issues are the collaboration problems that may have occurred between separate developer teams of Chrome. The team issues listed below were based on the conceptual architecture discussed in this report, under the assumption that this was the conceptual architecture used during development.

One team issue the developer teams of the Browser and the Renderers could have faced is the implementation of the multi-process architecture of Chrome. Google originally planned to create Chrome with one Browser Engine, and a single Rendering Engine consisting of multiple threads (Google Chrome Team, 2008). It was later decided that implementing a multi-process architecture would be more beneficial in order to sandbox each Renderer to block the transfer of bugs between tabs, and to prevent the entire Browser from crashing if one tab crashed. The Browser Engine developer team and the Renderer developer team would have to collaborate to design a system where the Browser could properly interact with numerous Renderers being created and destroyed. Significant collaboration between teams would be needed to ensure that the Browser subsystem could effectively communicate with multiple Renderers simultaneously.
Another team issue that could have occurred relates to the implementation of the Plugins system. As mentioned previously, both the Browser Engine and the Renderers are dependent on the Plugins subsystem. These two subsystems are affected by different plugins than the other, but both depend on the same Plugins subsystem to provide them with proper information and services. These three teams would have to collaborate to ensure the Browser and Renderer subsystems can both make different types of requests to the Plugins subsystem, and the Plugins subsystem is able to differentiate between the two types and provide appropriate services to both.

8. Conclusion

The conceptual architecture of Google Chrome was deduced as an object-oriented style architecture that involved five subsystems. The Rendering Engine subsystem employs a pipe-and-filter architecture style, and is a multi-process engine in Chrome, making it a unique facet of the browser. This, combined with the multi-thread approach renders a reliable and fast user browser experience. Future steps for the project involve delving further into the actual code that underlines the browser, and unveiling the actual architecture of the browser. This includes, but is

not limited to, the specifics of how the subsystems perform their functions in actuality, and features that the browser has (like security, and personalization).

9. Limitations & Lessons Learned

When arriving at the conceptual architecture of Chrome, there were various limitations that arose both during the process and in the architecture itself after it was established. However, it was the amalgamation of these limitations that allowed for various lessons that were learned as a result.

The research conducted to arrive at the conceptual architecture of Chrome involved various sources, such as primary articles, informative videos and professionally written blogs. However, it was found that there were varying opinions regarding the architecture of Chrome, and the connectivities that exist to allow the browser to function as it does. Therefore, the research conducted was authenticated with the information present on The Chromium Project. This source provides in-depth information regarding the various developments and updates of Google Chrome features. This can be considered a limitation in the number of available authentic resources, since this, and the videos from the development team, are some of the only resources that can provide evidenced facts. This limitation provided the grounds for developing the skills that are necessary to analytically evaluate the information presented, and judge its truthfulness, given the context. The group was able to individually gather information, bring it together, and discuss. From these discussions arose a deeper understanding of the system and subsystems that Google Chrome employs, and the deduction of their subsequent connectivities.

Using The Chromium Project as the most authentic source of information proved quite difficult, given that the documentation was very complicated and specific. The Chromium Blog is written for an audience consisting of developers that work closely with altering the low-level design of Chrome. The specificity of the information was such that it spoke to the individual processes of the subsystems, and their unique features that are modifiable. The overall connectivities and patterns of how the subsystems work together needed to be deduced. Doing so allowed the group to appreciate the vitality of unambiguous and traceable documentation, from both the perspective of the development team, and the consumer of the software.

The amount of information and opinions regarding the architectural structure of Google Chrome was found to be vast. It was discovered that there are varying opinions present even among professionals within the field. However, this taught the group the ability to filter relevant information.

Finally, though the process of understanding the architecture of Chrome, the effect that non-functional requirements can have on the layout of a browser was made apparent. Two browsers that perform the same function can do so very differently, and look very differently, due to the non-functional requirements that are deemed important.

10. Glossary

CSS - Cascading Style Sheets: Describes how HTML elements are to be displayed on screen.
CSSOM - Cascading Style Sheet Object Model: A set of application program interfaces allowing the manipulation of CSS from JavaScript.
DOM - Document Object Model: Programming interface that treats the makeup of websites as a tree structure where each node is an object representing a part of the document.
I/O - Input/Output
IPC - Inter Process Communication: The communication channel that relays messages between processes.
HTML - HyperText Markup Language: Standard markup language for creating web pages and web applications.
HTTP - HyperText Transfer Protocol: Protocol used by the World Wide Web to define how messages are formatted and transmitted.
XML - eXtensible Markup Language: Markup language with a strict set of rules for encoding documents so that it is human and machine-readable.
UI - User Interface: Where the user interacts with the browser.
URL - Uniform Resource Locator: Reference to a web resource that specifies where to find it on a web server and how to read it.

11. References

Chintada, S. (2017, December). Understanding Chrome Extensions : Part 1 (chrome architecture overview). Retrieved from https://medium.com/@saikumarchintada/understanding-chrome-extensions-part-1-chrome-architecture-overview-ef6abd7ff4aa

Crawford, S. (2011, August). How HTML5 Works. Retrieved from https://computer.howstuffworks.com/html-five.htm

DroidHead. (2017, December). Understanding How the Chrome V8 Engine Translates JavaScript into Machine Code. Retrieved from https://medium.freecodecamp.org/understanding-the-core-of-nodejs-the-powerful-chrome-v8-engine-79e7eb8af964

Garsiel, T., Irish, P. (2011, August). How browsers work: behind the scenes of modern web browsers. Retrieved from https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/#The_rendering_engine

Google Chrome Team. (2008). Behind the open source browser project. Retrieved from https://www.google.com/googlebooks/chrome/big_00.html

Grigorik, I. (2013, January). Constructing the Object Model | Web Fundamentals. Retrieved from https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model

Grigorik, I. (2013, January). High Performance Networking in Google Chrome. Retrieved from https://www.igvita.com/posa/high-performance-networking-in-google-chrome/

Grigorik, I. (2013, January). Render-tree Construction, Layout, and Paint | Web Fundamentals. Retrieved from https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction

Grosskurth, A., Godfrey, M.W. (2007). A case study in architecture analysis: the evolution of the modern web browser. Retrieved from http://cs.queensu.ca/~ahmed/home/teaching/CISC322/F18/files/emse-browserRefArch.pdf

Hassan, A.E. (Retrieved 2018, October). Software Architecture: Intro and Styles [Powerpoint Slides]. Retrieved from http://cs.queensu.ca/~ahmed/home/teaching/CISC322/F18/slides/CISC322_03_ArchitectureStyles.pdf

Kosaka, M. (2018, September). Inside Look at Modern Web Browser (part 1). Retrieved from https://developers.google.com/web/updates/2018/09/inside-browser-part1

Kurtuldu, M. (2018, September). How we designed Chrome 10 years ago. Retrieved from https://blog.chromium.org/2018/09/how-we-designed-chrome-10-years-ago.html

OpenHub.net. (Retrieved 2018, October). Chromium (Google Chrome). Retrieved from https://www.openhub.net/p/chrome/analyses/latest/languages_summary

Reis, C. (2008, September). Multi-process Architecture. Retrieved from https://blog.chromium.org/2008/09/multi-process-architecture.html

The Chromium Project. (Retrieved 2018, October). The Rendering Critical Path. Retrieved from https://www.chromium.org/developers/the-rendering-critical-path

The Chromium Project. (Retrieved 2018, October). Inter-process Communication (IPC). Retrieved from https://www.chromium.org/developers/design-documents/inter-process-communication

The Chromium Project. (Retrieved 2018, October). Threading and Tasks in Chrome. Retrieved from https://chromium.googlesource.com/chromium/src/+/lkgr/docs/threading_and_tasks.md