The Conceptual Architecture of Google Chrome

Assignment 1
October 19, 2018 (Fall 2018)
Thick Glitches

Alastair Lewis (15ahl1@queensu.ca)
Andrea Perera-Ortega (15apo@queensu.ca)
Brendan Kolisnik (15bak2@queensu.ca)
Jessica Dassanayake (15jdd1@queensu.ca)
Liam Walsh (15lcw1@queensu.ca)
Tyler Mainguy (16tsm@queensu.ca)

Abstract

A conceptual architecture for the Google Chrome web browser was derived through analysis of Chromium, an open-source version of Google Chrome. The intent when determining the final conceptual architecture was to create a model that valued security, speed, performance, and reliability. Architecture styles, coupling, and cohesion were heavily considered when choosing what subsystems to include.

The conceptual architecture was a combination of the layered architecture style and object-oriented architecture style. The subsystems that formed the architecture were the browser, network, plugins, rendering engine, and user interface. The goal was to choose subsystems and create dependencies that would result in low coupling and high cohesion. A significant reason as to why Google Chrome is widely used is that its multi-process architecture allows for concurrency of multiple processes. Along with an in-depth look at the conceptual architecture, use cases are presented in this report to demonstrate the interactions between the subsystems.

Through creating the conceptual architecture, a base now exists to allow the team to determine Google Chrome's concrete architecture. This base was formed by identifying the necessary subsystems and acknowledging the main dependencies of Google Chrome.

Introduction and Overview

Google Chrome, often referred to as Chrome, is a web browser that was released in 2008 for multiple platforms. Developed by Google LLC, its initial launch was compatible with Microsoft Windows, but adapted to be used on macOS, Linux, iOS, and Android at a later date [3]. Google Chrome has the majority of the market share for web browsers, with 67.1% of the market worldwide for January 2018 to October 2018 [2]. The browser entered the market at a time where internet users were demanding more of their time with browsers. Popular applications and video content required more than traditional static web pages. Google Chrome was able to facilitate a more dynamic experience for its everyday user by allowing more than one process to run at once and for plugins to be used on web pages for things like Adobe Flash Player and JavaScript. Google Chrome's multi-process architecture has allowed it to be one of the most stable and high-performance browsers.

There are several factors that have contributed to Chrome's popularity and they all arise from the web browser's architecture. By determining the conceptual architecture of Google Chrome, it was clear through the subsystems and their dependencies how the factors have led to success for the web browser. In order to derive the conceptual architecture, documentation for Chromium was used in place of Google Chrome's documentation because Chromium is an open-source browser. The main subsystems in the conceptual architecture are the browser, user interface, networking stack, plugins, and rendering engine. The dependencies between these subsystems formed an architecture that exhibits layered and object-oriented architecture styles.

Security, performance, reliability, and speed are some of the benefits of the Google Chrome web browser, and there are multiple reasons behind each benefit. For example, Chrome is highly secure because of the layered architecture and modular nature of the system. The dependencies between the system were chosen in a way that separated concerns and reflected actual connections to allow for low coupling and high cohesion. Coupling and cohesion come into play with Chrome's ability to handle concurrency and be reliable. Multiple processes are able to run at the same time and communicate with each other using Inter-Process Communication (IPC) as a result of the web browser's multi-process architecture [4], setting it apart from other browsers of its time, which historically only used single processes. The multi-process architecture uses multi-core CPUs, which contributes to the browser's speed. Google Chrome's innovative architecture structure is a significant reason as to why it has such a large share of its market.

Conceptual Architecture

Derivation Process

The initial step for constructing this architecture was understanding the domain extensively. This was achieved by first understanding general web functionality, such as servers, HTTP requests, and other web basics. Understanding the general structure of how web browsers have been constructed historically through reference architectures was also important in generating a base conceptual architecture. The logical next step for deriving Chrome's architecture was to discover how Chrome implemented aspects of the reference architecture, and how it differentiated from the reference architecture (whether extra components exist, or certain components are missing). After factoring in these steps, we found 3 major guiding factors to the conceptual architecture we derived: architecture styles, multi-process architectures, and coupling/cohesion. The understanding of a system's functionality is fundamental to guiding what architecture styles were used to implement them. This leads way to a mix between an object-oriented and a layered design. Multi-process architecture is the differentiating feature behind Chrome, so understanding the functionality behind it was crucial in understanding what components needed to exist, and how they communicated with one another. Tangential to multi-process architecture is coupling and cohesion. Understanding how components were segmented in order to facilitate multi-process architecture helps to understand their cohesive properties. A corollary to this being that communication between segmented components, running as separate processes means there needs to be a communication network (coupling) between sets of components. After all was considered, we were able to construct a conceptual architecture that reflected our findings.

Alternative Conceptual Architecture

Several alternative conceptual architectures were considered when deriving the final conceptual architecture. Architecture styles, multi-process architectures, coupling, and cohesion were the factors considered when determining what subsystems and dependencies made sense for the system. For this particular alternative, there were several reasons why this version was not used. For the render process and plugins process, each process is an instance of the rendering engine and plugin, respectively. In addition, we included networking in our final conceptual architecture, but did not include it in the alternative version. In the alternative version, it was intended that networking would be part of the browser process. However, these two subsystems were separated in the final architecture in order to increase cohesion. Ultimately, the alternative architecture was too concrete, which is why a different model was chosen.

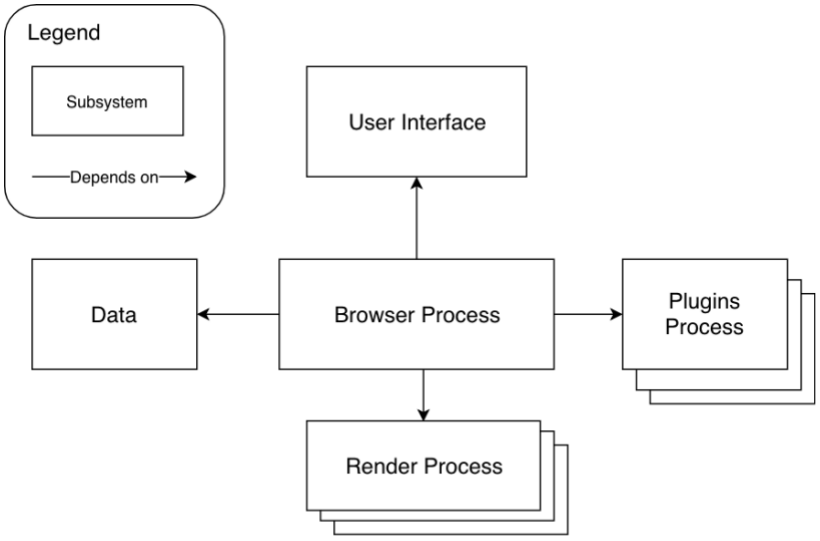

Figure 1. Alternative conceptual architecture

Final Conceptual Architecture

The final conceptual architecture that was developed can be seen in Figure 2. The system consists of five separate subsystems that operate independently from one another. Interactions occur between these subsystems, which are denoted in Figure 2 by arrows that represent the dependencies. The subsystems are the browser, UI, plugins, networking, and rendering engine.

The system as a whole is a mix of two architecture styles: layered and object-oriented. A layered architecture fits Chrome's functionality very well. The layered architecture style is visible with the UI acting as the top layer. The middle layer follows with the browser, plugins, and networking. Finally, the rendering engine is the bottom layer of the architecture. Each subsystem is a service to subsystem(s) in the layer above it. The layered style is beneficial to the architecture as it increases the security between the subsystems and also encourages modularity. Given the rapid development of the web, interchangeability and growth of components in Chrome is crucial. This is easily verifiable by reviewing the history of Chrome's rendering engine, which replaced the Webkit rendering engine for their own "Blink" rendering engine.

Due to the multi-process architecture used in Chrome, objects (separate processes for example) must be able to reference one another in order to transfer data over communication interfaces. Object-oriented structuring gives way to abstract of components, which allows for various implementation details to be hidden from other components (a beneficial security measure). An extension to the benefits includes the autonomy of components; concurrency of rendering and browser processes are direct beneficiaries of this.

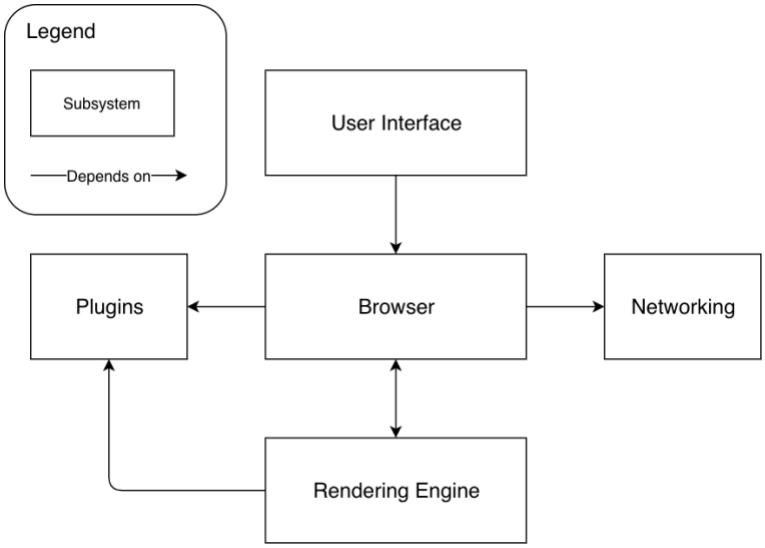

Figure 2. Conceptual architecture of Chrome

Subsystems

User Interface

The user interface subsystem is the link between the browser and the users themselves. It is the first layer in the conceptual architecture and handles all forms of user input. It's the graphical display that reflects the state of the system as a whole in real-time. It provides users with the web navigation experience they are familiar with.

The browser subsystem receives bitmaps of web pages from the rendering engine, and hands them off to the UI to be displayed within the viewing pane. The user interface is in constant communication with the browser subsystem. Every time a user so much as moves their mouse, the user interface will alert the browser subsystem. Earlier web browsers would have the UI communicate directly with the rendering engine. But this was a much less secure way of operating.

The user interface depends only on the browser subsystem.

Browser

The browser subsystem is the central hub system of Chrome. This subsystem manages other components of the application and their permissions. The browser interacts with the operating system and contains persistent storage this includes the cookie database, password database, history database as well as disk cache. The browser communicates with the networking subsystem using IPC and thus has a download manager subsystem. For security, many of the other subsystems do not have direct access to each other and instead must facilitate through the browser subsystem as the central system. Thus, the rendering engine cannot directly access networking or the filesystem if it is compromised. This also applies to plugins (such as Flash). This increases security in case the rendering engine or plugins are compromised as the browser manages info on what privileges it has given each rendering engine process (as well as plugins).

The browser implements the tab-based windowing system (including the URL bar) and is the intermediary between the UI (which is OS specific) and rendering engine [6]. As the browser is the only subsystem with access to networking when a user wants to download a file or go to a site it is the browser that communicates with the networking subsystem receives the response and passes it to the rendering engine. The browser does not decode or run any client-side code for security purposes and hands it straight to a rendering engine sandbox process [6]. The developers were careful so as to minimize the risk to the browser because of its access to the disk and networking subsystem. If a file was to be downloaded the rendering engine would pass back the file after unzipping and where it should be saved. Thus, the browser is in control and can determine if the file is safe.

The browser subsystem handles user input events it receives from the operating system and if the input event is from the content area it is then passed to the rendering engine. Only the input events intended for its content area can be viewed by a compromised rendering engine for additional security [6].

Lastly, the rendering engine does not have direct access to the UI but sends the bitmap to the browser subsystem which then displays it in the UI [6]. This is for additional security, to decrease coupling and possesses only a small performance penalty. To facilitate these design choices the browser subsystem is dependent on the rendering engine, networking, and plugins. The browser is the most critical subsystem in terms of security which is why so much effort is put into preventing a crash and protecting it from compromise as it has access to all other subsystems.

Networking

The networking subsystem handles all communication that Chrome has with the external internet for a variety of resource fetching. Any request relative to loading data into the browser must be communicated through the networking subsystem. The general service provided by the networking subsystem is to facilitate HTTP requests made by the requesting components [8].

The responsibility of the networking subsystem is also to correctly parse retrieved data into a data type that can be received calling components. Additionally, the networking subsystem will deal with caching data from network requests [9][10]. This sort of functionality is widespread in browser networking subsystems, in order to reduce redundant network calls (for both efficiency

and keeping networking threads from becoming more congested). The networking subsystem will also deal with the cookies associated with various requests [11]. The job of the networking subsystem is strictly to identify relevant information of cookie data being passed through it (whether by client request or server response), and should not be concerned with direct storage details [11].

The networking subsystem does not depend on any other of the major subsystems in the architecture, as its concerns are facilitating requests made by other components who require access to external data. By restricting the communication from other components (i.e. rendering engine/plugin processes), it allows for the concern of permissions of requests to be handled outside of the networking subsystem. This increases the cohesion of the networking subsystem, as it remains only concerned with what requests to facilitate, without having to worry about the validity of the calling process [6].

As a result of the networking subsystem being the only access-point for the external internet, it is important that none of the processes that are attempting to access queried data interfere with one another. This means that queries made by the networking subsystem must be asynchronous, as a means of facilitating incoming requests while waiting for data to be returned [8].

Rendering Engine

The rendering engine exists to fill each tab with web content in the viewing pane. The subsystem makes use of Blink, a fork of WebKit [14], to facilitate the rendering process. Each tab runs its own instance of the rendering engine. The subsystem as a whole is sandboxed with no access to the computer's disk or the network stack. These restrictions ensure that a compromised instance of the rendering engine will not cause any harm to the user's computer, nor will be able to send compromised data out to the internet.

Within its rendering engine module, Chrome includes a powerful JavaScript engine that employs a different technique for handling the source code. Rather than interpreting JavaScript, it will compile the code and execute it afterward, as needed [15].

The rendering engine will receive an HTTP response from the browser, containing content to be rendered. Then, the instance of the rendering engine associated with the corresponding tab will begin to parse the data in the body of the response. Any content that requires a plugin to be loaded, such as an adobe flash applet, will be sent to the plugins subsystem for processing and returned in a format that the rendering engine can generate a bitmap for. After the browser subsystem receives the bitmap from the rendering engine, it will paint it into the tab's display window in the user interface.

Web pages have become so complex and so large nowadays, that it would be impossible to render them all in the same process. Chrome's multi-process architecture allows each instance of the rendering engine in chrome to run asynchronously in their own separate sandboxed process. This separation and sandboxing will not only accommodate the large and complex pages, by giving them dedicated resources, but it will ensure that one process's failure does not interfere with the rest of the system. If an instance were to crash, only the tab associated with it would

become unresponsive. If an instance were to be compromised, its isolation from the other instances would not allow it to access them [16]. However, since these processes are split into separate instances, this means more RAM usage for the computer.

The rendering engine subsystem is very performance critical; it is where most of the hard work is done. It takes a lot of a computer's resources to render bitmaps in real time, and therefore a lot of work was put into optimizing everything related to the rendering engine. The multi-process architecture plays a huge role in allowing the rendering engine to run smoothly, it takes full advantage of multi-core CPUs, which are becoming the standard nowadays.

The rendering engine depends on the browser subsystem and the plugins subsystem. It relies on the browser to supply it with web content to render and to resolve any disk or network requests. It depends on the plugins subsystem to assist it in rendering content that is outside its scope, its direct communication line with the plugins subsystem increases the speed at which it can render a page in full.

Plugins

Chrome supports plugins like other browsers to add additional features to the browser (typically made by third-parties). The plugins subsystem represents the plugins processes. This subsystem is independent of the rendering engine as there is at most one instance of each plugin rather than having one per rendering engine process [6]. Had the plugins been included in the rendering engine they would consume lots of memory and potentially break the sandbox design if the plugins are not sandboxed. Additionally, the plugins must be independent of the browser subsystem as if a plugin crashes the whole browser would crash [6].

Plugins such as Flash are required to be sandboxed now with no direct access to disk and there are settings to require all plugins to be sandboxed in order to run. Plugins run in their own independent process and communicate with both the rendering engine and the browser subsystem [6]. When the last tab using a plugin is deleted the plugin instance is deleted. Since plugin content is rendered directly from rendering and the two subsystems communicate over IPC both ways, the plugins subsystem is an independent system and not dependent on anything.

Concurrency

The purpose of constructing a browser using a multi-process architecture is to achieve concurrency of processes. This means allowing processes to run in parallel, where their execution (generally) does not concern itself with the outcome of other processes. Traditionally, browsers ran in a single process in order to reduce the client's memory usage. As the web became more complex and web applications became more commonplace, a single process architecture didn't scale to meet its demand [13].

The introduction of the multi-process architecture in Chrome allowed for complex calculations, network requests, and more to be run in parallel to one another. The primary purpose of running these processes concurrently was in order to increase the speed at which web pages could

operate. Being able to execute various aspects of loading a web page (HTTP request, compile JavaScript, etc.) in parallel means that more can be achieved in a given time, combining to create an identical end result [4].

The other main benefit behind implementing a multi-process architecture is removing single-points of failure. A single processor implementation of a web browser can easily be hung or crash in the situation where a malicious site or a misbehaving plugin is present [4]. By sandboxing these elements into distinct processes, it prevents a single process from influencing other processes occurring in the browser. Additionally, running concurrent processes helps isolate data from other components (as there's no direct communication between most processes) [6].

The tradeoff behind running concurrent processes is an increased memory usage [12]. Each separate process that must be spun up takes a chunk of the client's available RAM. As the number of plugins and tabs increases, the relative RAM Chrome consumes grows to quite large values. If a client's system has a lower end quantity of RAM, the execution of Chrome can actually slow down overall CPU performance.

How the Architecture Supports Future System Changes

The architecture style is OOP and layered so one advantage is that implementations of systems can be changed without affecting the overall system. For example, changing the implementation of the rendering engine would not affect the UI system. The IPC cannot be easily changed as it is tightly coupled into multiple systems including the rendering engine, plugins, and the browser system. This architecture is very extensible. New systems can be added to a layer without changing the implementations or communication systems of other systems. Chromium is the open source code that Chrome is based on which allows the community to contribute new features that can be implemented into production Chrome. Leveraging this strategy, the Chrome team can get the community to implement many of the new features going forward (while monitoring for performance and security).

External Interfaces

There are three different external interfaces that Chrome communicates with. The first interface is the GUI which allows the browser to process user input events and send visual output to the user by utilizing the User Interface system. The second interface is the file system which allows Chrome to interact with the local files stored on the host machine. Chrome is limited to accessing the files that the operating system has given it permission to read and write to. Finally, the browser is able to interface with the network through the networking stack which allows it to send and receive content that is not stored locally.

Use Cases

Downloading and Displaying a Web Page

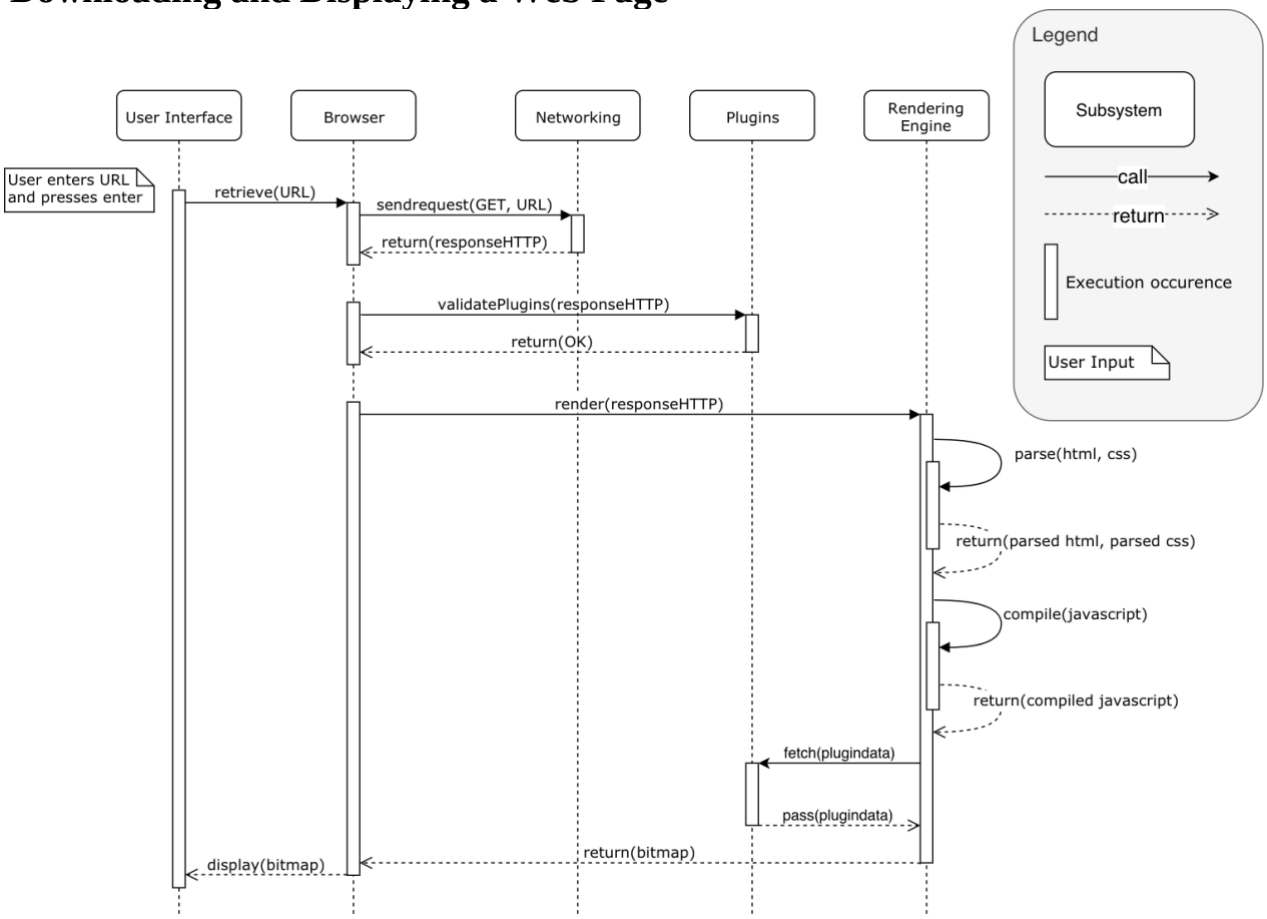

Figure 3. Sequence diagram of downloading and displaying a web page

The first use case illustrates the process that happens within Chrome during an ordinary fetch and display of a webpage. The process begins with the user inputting their desired URL into the search bar in the User-Interface and pressing the enter key. Upon the enter key being pressed, the UI will pass the URL string to the browser where it is wrapped in one of various request types such as GET, POST or PUT. The request is then forwarded to the network stack where it is sent out to the network and the program awaits a response. When the HTTP response is returned to the network stack, it is forwarded back to the browser for processing.

The browser takes the HTTP response and unpacks the various components of it such as the HTML, JavaScript and CSS and checks the code to see which plugins are necessary to display all of the content. The Browser will then query the Plugins module and confirm that it has all the necessary software to properly load the webpage, upon confirmation, the browser will then pass the HTML, CSS, and JavaScript to the rendering engine.

The Rendering Engine then interprets the HTML and CSS and generates the layout and design of the webpage. The JavaScript is then compiled (as V8 compiles to bytecode rather than interpret line by line) (add citation here) and added to the webpage's design. The plugins module is then called on to add any other components into the webpage that require third-party software stored in the module. Once everything has been processed and the final webpage layout and design has been finalized, the generated bitmap is sent back to the browser.
The browser takes the bitmap and sends it to the User Interface where it can be displayed onto the screen for the user to see and interact with.

User Logs into Website and Chrome Saves Username and Password

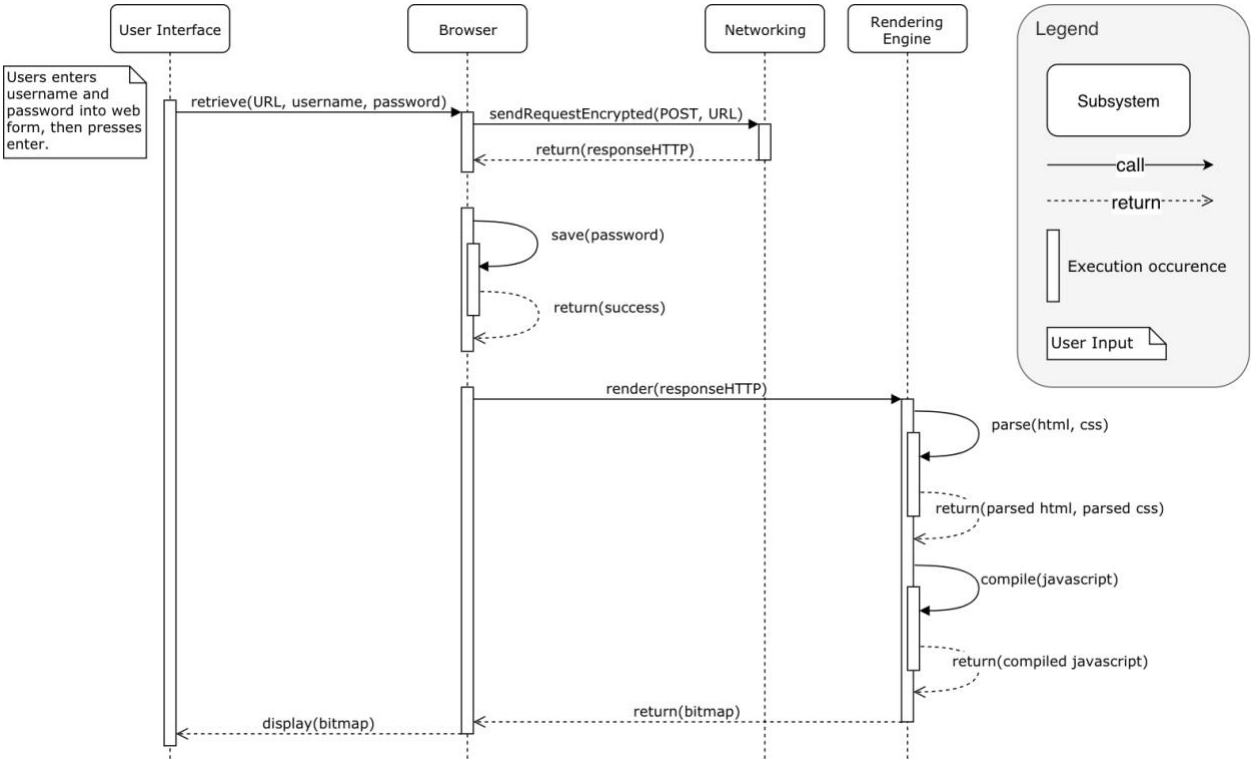

Figure 4. Sequence diagram: logging into a website; storing the password in persistent data

The second use case illustrates the process that happens within Chrome during a successful login to a website and the storage of the username and password. The process begins with the user inputting their credentials into an HTML form in the User-Interface and pressing the enter key.

Upon the enter key being pressed, the User-Interface will pass the URL string and form data to the browser where it is wrapped in a POST request. The request is then forwarded to the network stack where it is sent out to the network using SSL encryption and the program awaits a response. When the HTTP response is returned to the network stack, it is forwarded back to the browser for processing.

If the login response was successful, then Chrome will take the username and password and store them in a local file on the host computer's disk.

Team Issues

It is inevitable that when dealing with an application of such magnitude, there are bound to be a multitude of challenges throughout the development process. When dealing with the architecture of Chrome, or any architecture in general, it is important to consider the dependencies between subsystems, as they can interfere with workflow. For example, the browser subsystem of Chrome possesses multiple dependencies on other subsystems, those being the rendering, plugins and networking subsystems. Because of this, the team in charge of the browser subsystem is required to interact more often with other teams, therefore making it less independent as the other teams. Due to the decreased independence of the browser subsystem team, more time is required to be spent discussing with the teams in charge of subsystems of which browser depends on, slowing down the development process and workflow.

Limitations and Lessons Learned

In regard to our own team, we experienced several limitations and challenges throughout the duration of the assignment that we had to overcome. The most difficult limitation that we encountered was related to gathering information and research. As a group, we noticed that there is a vast amount of new and old information available on the internet. This can make it overwhelming to obtain valuable information, especially because a lot of us did not have pre-existing knowledge on how Google Chrome works. Technology is constantly evolving and improving, so it was difficult to ensure that what we were learning was up to date and still used by Chrome. A specific example of this being that most architecture documentation of Chrome is from its release in 2008, and yet to be updated. An additional limitation we encountered is the fact that Chrome is a commercial product and not open source. This forced us to look at Chromium, which is open source.

Despite facing various limitations during the course of the assignment, we were also fortunately able to learn valuable lessons. Group size was initially a limitation for us, due to busy schedules interfering with the meeting process. However, we were able to overcome this by communicating a lot as a team both online and in person. We organized meetings in real life as much as possible, even if it was for a short amount of time, or if several members could not attend that day. We felt that any time we were able to dedicate to meet was valuable. In spite of this, the benefit of working with a team outweighed the difficulty that came with it. Being in a group setting made it considerably easier to discuss and develop ideas for developing the conceptual architecture. Specifically, working in a positive environment with other hardworking individuals made the brainstorming process a lot more pleasant, as we were able to participate in valuable and productive discussion, without fear of being judged for proposing a different idea, or saying something incorrect.

Conclusions

In summary, the derived conceptual architecture of Google Chrome is object-oriented and layered. The use of these architecture styles was able to contribute to several benefits that have made Chrome so popular. The web browser has a heavy focus on security and this is implemented through the architecture styles that allow subsystems to be independent. Furthermore, the performance and reliability of the system are enhanced through the system's design that values low coupling and high cohesion. This was done by dividing the requirements of the architecture between the subsystems and forming interactions through dependencies where necessary. Finally, Google Chrome's use of a multi-process architecture has allowed processes to run concurrently, which increases its speed and makes the web browser much more attractive to users looking for high speeds when browsing.

Through determining Google Chrome's conceptual architecture, a foundation is formed that can be used to derive the web browser's concrete architecture. This is beneficial because concrete architectures provide a more in-depth look into the system and can provide more detail about interactions between different components than a conceptual architecture.

Data Dictionary/Naming Conventions

Blink: The rendering engine used by Google Chrome in their current architecture. Blink is based on the WebKit rendering engine used in previous versions of Chrome.
Bitmap: A mapped output generated by the rendering engine to be displayed in the User Interface graphically
Browser: The main module in the program that controls data exchange across modules in the application. It is the only module in the architecture that directly interacts with the operating system in order to have functionality for data persistence and resource allocation.
Cohesion: The level of separation of functionality between modules in the software. A highly cohesive system will have modules that perform very specific tasks tailored to their attributes.
Concurrency: Multiple software processes running at the same time by utilizing multiple processing cores.
Coupling: The number of dependencies your program has in between its various modules. It is optimal to have low coupling in your system so that if one module malfunctions there is minimal effect on the other modules in the architecture.
CSS: (Cascading Style Sheets) Lightweight coding language that directly interacts with HTML to allow webpages to have custom appearances and layouts.
Data Persistence: The ability for an application to access the host machines file system in order to read and write to files. It allows for the storage of various types of data such as history, bookmarks, and passwords so that when the application is closed and reopened it still has data stored from the previous browsing session.
GET Request: A request that is sent out into the internet with relevant data for the query stored in the request header.
HTML: (Hypertext Markup Language) The format in which web pages are written in. Uses text with 'tags' surrounding them to describe how they should look and appear on the page.
HTTP: (Hypertext Transfer Protocol) The protocol used by the internet to send HTML across the web from servers to clients.

IPC: (Interprocess communication) Set of programming interfaces that allow for communication and coordination between software processes running concurrently.
JavaScript: An event driven programming language that shares similar syntax to Java that allows web pages to be interactive and have features that would otherwise be impossible with only HTML.
Layered Architecture: Architecture style with multiple levels to it where data moves from one processing level to the next. Each layer is a server to the layer above it and a client to the layer below it.
Modularity: The process of subdividing software into individual components. It results in more understandable code and allows for changes to individual modules without having it affect all others.
Multi-Processing: Software that is designed and optimized to run on computers with multiple processing cores.
Network Stack: The module in the web browser that is responsible for actually interacting with the internet in order to send and query for data in the world wide web. This allows the browser to collect data that is not stored locally on the host machine.
Object-Oriented Architecture: Architecture style with multiple, single-purpose 'Objects' that have interfaces to interact with other objects in the system. Objects are not concerned with the implementation of the other objects in the system, only with the data getting passed to them.
Plugins: Third-party software that is not included in the application by default. They add extra functionality to the software that the user can take advantage of and get a more customized browsing experience.
POST Request: A request that is send out into the internet with relevant data for the query stored in the request body as opposed to the header
Rendering Engine: Large piece of software that is capable of taking in HTML, CSS, JavaScript, and other languages in order to parse and compile them together into a graphical bitmap page that can be interpreted and displayed to the user in the User Interface.
Sandbox: The programming practice that segregates a certain set of functionalities to a restricted amount of resources, file access, and operating system interactions on the computer in order to increase security. This way, it the sandboxed portion of the application crashes or is compromised, it will have no serious effect on the rest of the system.
UI: (User Interface) The graphical representation of the application that interacts with the user to show/get input and output from them.
URL: (Uniform Resource Locator) A string of text that identifies a specific webpage on the world wide web. These are used by the web browser to retrieve specific pages requested by the user.

References

[1] “Blink.” *The Chromium Projects*, www.chromium.org/blink.

[2] “Desktop Browser Market Share Worldwide.” *StatCounter Global Stats*, gs.statcounter.com/browser-market-share/desktop/worldwide/#monthly-201801-201810-bar.

[3] “Google Chrome.” Wikipedia, Wikimedia Foundation, 16 Oct. 2018, en.wikipedia.org/wiki/Google_Chrome.

[4] “Inter-Process Communication (IPC).” *The Chromium Projects*, www.chromium.org/developers/design-documents/inter-process-communication.

[5] “Multi-Process Architecture.” *The Chromium Projects*, www.chromium.org/developers/design-documents/multi-process-architecture.

[6] “The Security Architecture of the Chromium Browser”. Barth, A., Jackson, C., Reis, C., & Google Chrome Team, 16 Oct, 2018. http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf

[7] “Multi-Process Architecture” *Chromium Blog*, https://blog.chromium.org/2008/09/multi-process-architecture.html

[8] “Network Stack”, *The Chromium Projects*, https://dev.chromium.org/developers/design-documents/network-stack

[9] “Disk Cache”, *The Chromium Projects* https://dev.chromium.org/developers/design-documents/network-stack/disk-cache

[10] “HTTP Cache”, *The Chromium Projects* https://dev.chromium.org/developers/design-documents/network-stack/http-cache

[11] “CookieMonster”, *The Chromium Projects* https://dev.chromium.org/developers/design-documents/network-stack/cookiemonster

[12] “Why is Chrome Using So Much RAM? (And How to Fix it Right Now). Albright, Dann, 17 June, 2017 https://lifehacker.com/why-chrome-uses-so-much-freaking-ram-1702537477

[13] “Explore the Magic Behind Google Chrome”. Zeng, Dico, 21 Mar, 2018 https://medium.com/@zicodeng/explore-the-magic-behind-google-chrome-c3563dbd2739

[14] “Which WebKit Revision Is Blink Forking from?” Google Groups, Google, 18 Apr. 2013, https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/J41PSKuMan0/gD5xcqicqP8J.

[15] “Understanding How the Chrome V8 Engine Translates JavaScript into Machine Code.” FreeCodeCamp.org, FreeCodeCamp.org, 20 Dec. 2017, https://medium.freecodecamp.org/understanding-the-core-of-nodejs-the-powerful-chrome-v8-engine-79e7eb8af964.

[16] Kosaka, Mariko. “Inside Look at Modern Web Browser (Part 1) | Web | Google Developers.” Google, Google, Sept. 2018, https://developers.google.com/web/updates/2018/09/inside-browser-part1.