Chrome: Conceptual Architecture

Assignment 1 – Report

Oct 19th, 2018

Mackenzie Furlong – 15mwf1@queensu.ca
Alex Golinescu – 16ag16@queensu.ca
David Haddad – 16dh10@queensu.ca
William Melanson-O'Neill – 16wmon@queensu.ca
Michael Reinhart – 15mr34@queensu.ca
Lianne Zelsman – 13lkz1@queensu.ca

Abstract

The team was to research the Google Chrome web browser and propose a conceptual architecture based on the studied findings. A layered architecture was initially proposed due to its benefits surrounding system component reuse and easy design evolution. The design was not chosen due to it ineffectual structure for Chrome's subsystem relationship complexity. Eventually an object-oriented design structure was selected.

The abstract architecture is comprised of five main subsystems: networking, rendering engine, user interface, data persistence and browser engine. The networking subsystem connects to the internet using FTP and HTTP. The rendering subsystem, made of multiple components, parses HTML and CSS and prepares the Document Object Model (DOM). The user interface presents the page and allows the user to interact with the browser. The data persistence subsystem collects and stores continuous data and information from users. The browser engine represents the top-level browser window and acts as the system's main control center. Each subsystem is a separate object in the object-oriented design structure, with various dependencies on each other.

The object-oriented design makes for easy scalability and evolution. New subsystems can be added and tested individually before being implemented into the preexisting system. The division of work between teams is simplified by each subsystem being its own object. Each development team can be given the ownership of a specific object, allowing them the autonomy to change the internals of the subsystem without affect other parts of the system.

Dependencies between objects can create complexity for development teams, as every dependent subsystem needs to know the state of all other subsystems. Object-oriented design can be less efficient than other architecture styles, using more CPU and memory on average. Even with the efficiency trade-off, the benefits the object-oriented structure provides to development teams are substantial.

The quality and thoroughness of the conceptual design were limited by time constraints, as well as the team's knowledge of web systems and resources available. Chrome's vast, complex architecture made it impossible to investigate each system component in detail, and research had to prioritize the main structure's functionality while omitting secondary features and extensions.

Table of Contents

Table of Figures

Introduction and Overview

This report discusses the conceptual architecture proposed by the team for Chrome—a web browser built by Google based on the open-source Chromium project. The design is based on—but not identical to—Chrome's existing architecture as described in the official Chromium Project documents, Chrome developer guide, Chromium and Google blogs, as well as other third-party articles and videos. The team took into consideration the various functional and non-functional requirements of Chrome when developing an architecture suitable to achieve the web browser's goals, choosing to use an object-oriented style for the overall structure.

The report discusses the design derivation process and alternative architecture ideas considered by the team, with focus on why the team initially considered a layered architecture design and eventually decided against it. The Architecture section provides details and explanations for the determined abstract architecture. Each major component is discussed, along with their interactions and data flow. The report covers various technical benefits—testability, scalability and evolution—of the determined architecture, and their implications for concurrency and software development. Design trade-offs, such as the performance concerns and structural difficulties, are also discussed.

The External Interfaces section provides details surrounding information transmitted to and from the system, with focus on plugins and how user data is obtained and stored in the cloud. Two use-cases, one for a user successfully logging into a webpage and another for the JavaScript rendering process, are walked through in the Use Cases section. Sequence diagrams are provided for clarification on how the major components interact with each other and the user. The Data Dictionary section has a glossary that briefly defines key terms, and the Naming Conventions section outlines the acronyms used throughout the report.

The Conclusions section discusses how the team came to realize problems caused by Chrome's complexity and size. However, the chosen object-oriented architecture brings benefits to security, availability, testability and scalability, each satisfying non-functional requirements (NFR) and simplifying the software development process for teams. The omission of extensions and web apps from the conceptual architecture is discussed and will be part of the team's focus moving forward. The Lessons Learned section focuses on the team's experience dealing with unsatisfactory resources for determining a high-level picture of Chrome's overall architecture. The differences between Chromium and Chrome are discussed, as well as their effect on the accuracy of outdated sources. As the team found that they rushed through the derivation stage of the architecture design, future design decisions will be researched and evaluated more thoroughly before a commitment is made.

References are provided at the end of the document. Readers looking for more details about the studied Chrome architecture may use the provided links to receive lower-level explanations from the Chromium Project documents.

Architecture

Derivation Process and Alternative Architectures

When deciding on a conceptual architecture to use for the browser's design, the team considered various architectural styles. One of the main investigated alternatives was a completely layered architecture. Having a layered architecture would make reuse and design evolution easy. Having closed layers would enforce low coupling of components, and changes to layers could happen in isolation without affecting other non-associated layers.

The team eventually decided against the layered architecture for a few reasons. It was determined that Chrome's vast system and complex component relationships made it difficult to structure the entire architecture in a layered way. As each layer can only interact with the one above or below, it must first travel through each layer in its path to reach a layer at a more distant level. With Chrome's high number of interdependencies between its components, it would be inefficient to require communication to move through unnecessary layers to complete a simple transaction. The team determined that this could greatly hinder performance, which is a key non-functional requirement of the web browser.

Instead, the team decided to go with an object-oriented design style for the overall system. Using an objected-oriented structure provides various benefits that will be discussed in the sections below. However, some individual objects will still use a layered architecture for their components, such as the application layers in the rendering engine subsystem.

Major Components and Interactions

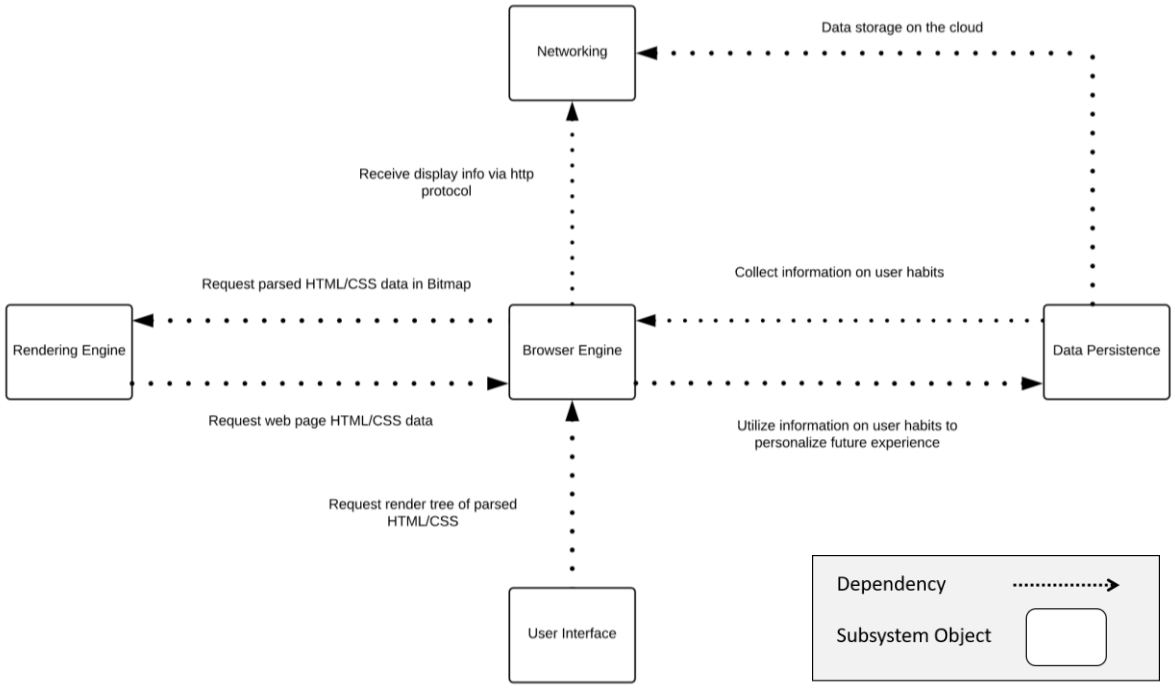

Figure 1: System Dependency Architecture Diagram

The team decided to divide the browser architecture into five components, each one organized as a separate object in the system, as seen in Figure 1 above:

- **Networking:** interacts with the rendering engine to connect to the internet using FTP and HTTP. It translates different character sets and implements a cache for retrieved resources.
- **Rendering Engine:** interacts with the networking system and browser engine to accept HTML and CSS and parses it into information that can be read by the UI (it prepares the Document Object Model (DOM)). Blink is used as the rendering engine.
- **User Interface (UI):** presents the page and its features; it is how the user interacts with the browser.
- **Data Persistence:** works with the browser engine to collect and record continuous data and information from users.
- **Browser Engine:** runs the UI and manages the tab and plugin processes. Delegates HTML and CSS code to be sent to the rendering engine(s) for parsing.

The browser and render processes work together with various subcomponents, each organized in a layered application structure consisting of Blink (Webkit based render engine), WebKit Port, WebKit Glue, Render Host, WebContents, Browser and Tab Helpers (all of which are defined in the Data Dictionary section) [1].

The Data Persistence subsystem is made up of various components:

- **Plugins**: information on what plugins are used by the user.
- **Chrome Generated User Model:** builds a model of the user through the collection and manipulation of their usage data. This model is stored in the cloud to be used by Chrome to tailor the user's experience and sold to third-parties for commercial purposes. Data analytic techniques are done to predict future behavior of the user.
- **Browser History:** keeps a record of previous search history of the user. This information is stored both locally in the browser, as well as in Google's cloud storage.
- **Cookies:** records personal information, such as auto-fill information for online forms. This information is stored in the browser, as well as in Google's cloud storage.
- **Cache:** files, images, and sites are saved in cache memory for faster retrieval (stored locally).
- **Extensions**: user installed extensions (stored locally).

The interactions between these components and other subsystems can be seen in Figure 2 below:

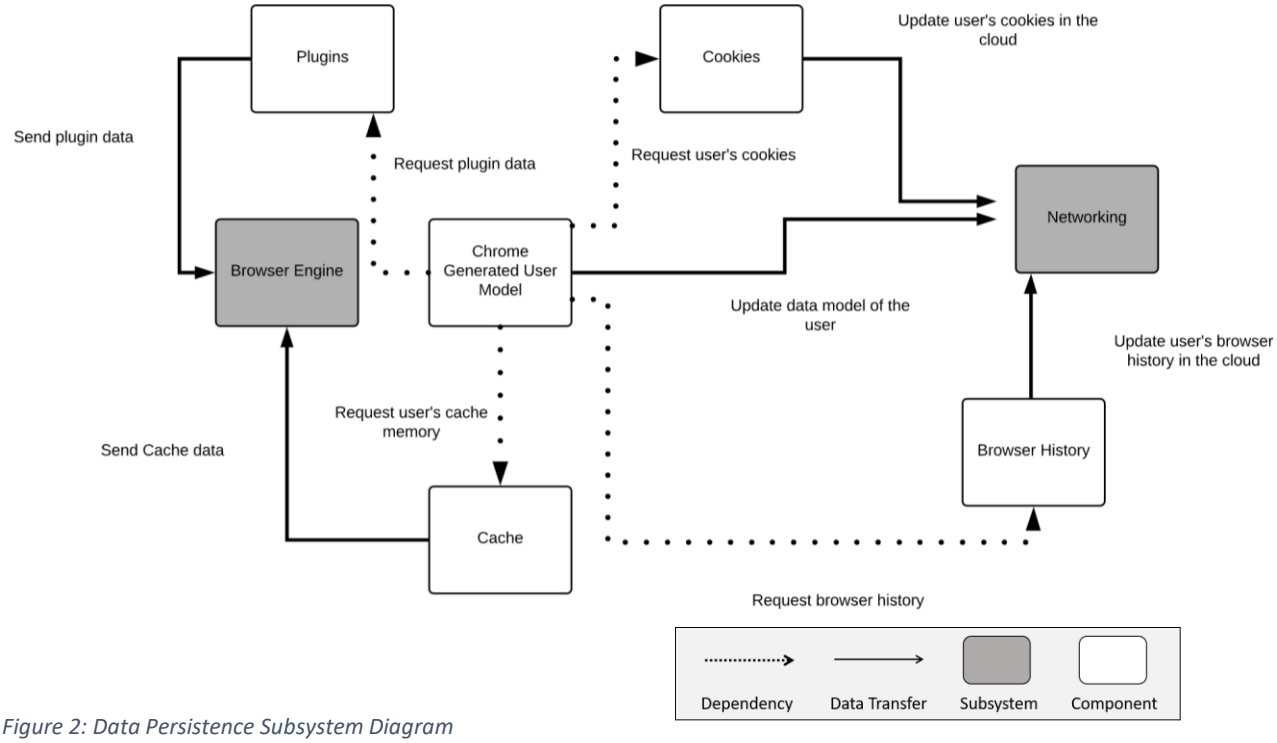

Figure 2: Data Persistence Subsystem Diagram

Concurrency and Data Flow

Each tab in Chrome runs its own instance of the rendering engine, which allows the tabs to operate independently and concurrently from one another. The browser engine manages all the render processes and displays the UI. It does this by maintaining two threads: the main/UI thread and the I/O thread [2]. Multiple render instances are run, splitting rendering into sections that do not have knowledge or dependencies on any higher-level layers.

As Chrome has a multi-process architecture, there are a large amount of processes that need to communicate with one another—e.g. the browser engine, renderer and plugins. This inter-process communication (IPC) is done via pipes. A pipe is allocated for each renderer process for communication with the browser process. Pipes are asynchronous to avoid blocking one another [2].

The Main/UI thread is responsible for rendering web pages on the screen, while the I/O thread takes care of IPC communication between the browser process and render process, as well as any network communication. Resource requests—e.g. for web pages – can be handled entirely on the I/O thread and do not block the user interface.

Render Process

On the Main/UI thread there is one *RenderProcess* object per render process [3]. There is also the *RenderView* object, which communicates with its corresponding *RenderViewHost* in the browser process and the renderer layer [1]. This object represents the contents of one web page in a tab or popup window. This process can be seen in Figure 3 below:

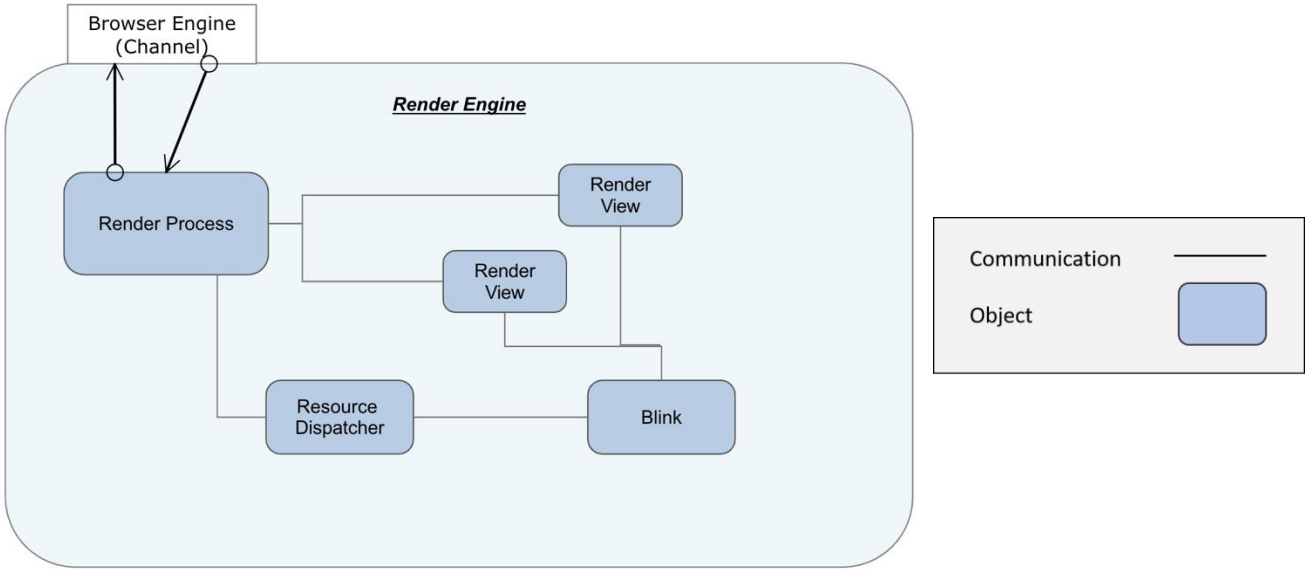

Figure 3: Render Engine Architecture Diagram

Browser Process

The Main/UI thread has the browser and its multiple *WebContents* for displaying each web page [1]. There is one *RenderProcessHost* connecting to each render process, which dispatches view-specific messages to the *RenderViewHost* [3]. The *RenderWidgetHost* handles the input and painting for *RenderWidget* in the browser [3].

The I/O thread has a *channel* which defines methods for communicating across pipes. There is a *ResourceDispatcherHost*, which sends network requests over the internet, and the *RenderProcessHost* object which receives the IPC requests from each renderer [1]. These messages are forwarded to the *ResourceDispatcherHost* [1]. This process can be seen in Figure 4 below:

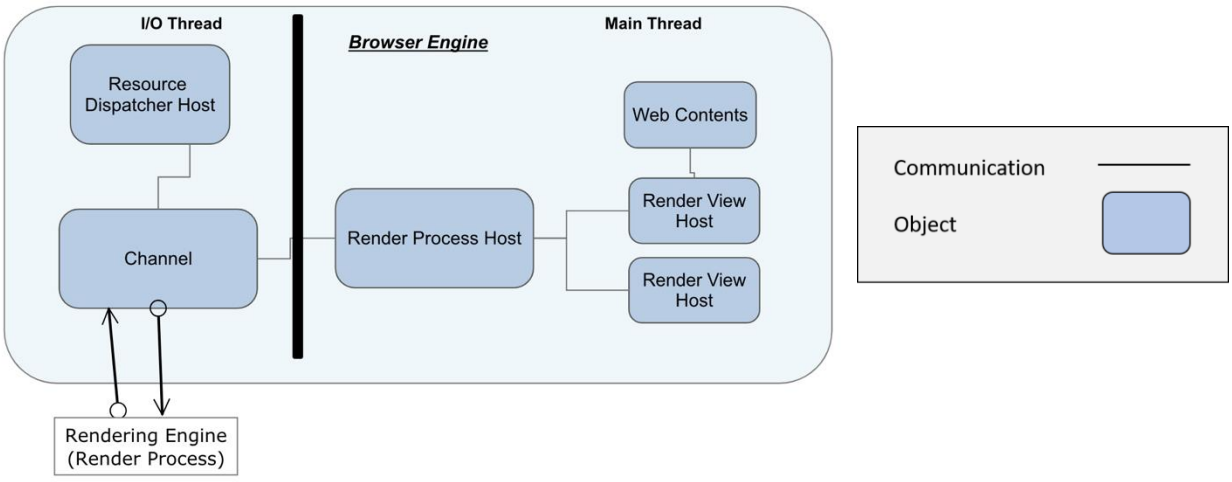

Figure 4: Browser Engine Architecture Diagram

One of the key non-functional requirements of a web browser is high availability, as issues and downtime can severely impact the success of the product. The concurrency provided by multiple rendering processes helps ensure availability in the browser, as errors in one process will not impact others.

The rendering engine is the most performance critical component in Chrome, as each tab contains its own instance of the engine (which all run in parallel). However, there is a limit to this behavior and it is dependent on the machine that the web browser is running on. Chrome intelligently assesses the computer and its performance when creating multiple rendering engines to decide on when to stop making each tab a separate rendering process. It will then fall back to the old style of consecutive processing and while it may appear as though processes in separate tabs are running simultaneously, they are in fact sharing a rendering engine. To run at optimal performance Chrome ideally attempts to not overload the computer and run into the situation just described. To do this, each render engine must run extremely efficiently.

Developer Implications and Design Tradeoffs

Benefits

The modularity of the individual objects provides building blocks for easy scalability and evolution. New components can be added and tested individually before being implemented into the preexisting system. The design style also makes the system easily testable. The functionality of each object can be tested individually before testing the relationships between objects and components, as well as their combined functionality.

The object-oriented structure also works to achieve one of Chrome's primary non-functional requirements of high security. Modularity allows the browser to easily separate functionality and sandbox individual processes.

Division of work between teams is simplified by each subsystem being its own object. Each development team can be given the ownership of a specific object, allowing them the autonomy to change the internals of the subsystem without affecting other parts of the system.

Issues

Dependencies between objects can create complexity for development teams. Developers must understand the way subsystems interface with each other and how changes to one object could affect a dependent object. This can make it difficult to change certain components, as every dependent subsystem needs to know the state of all other subsystems.

Object-oriented design can be less efficient than other architecture styles. On average, object-oriented code uses more CPU and memory than procedural code. This means that it is essential for development teams to design code systems in a way that helps optimize efficiency.

External Interfaces

Plugin Architecture

Web apps are designed to be run independently of each other in the browser; they can be run in parallel. The same is true for browser plugins—such as Flash—which are loosely coupled with the browser and can be separated from it with ease. Plugins can cause browser instability as they make sandboxing impractical since they are third-party programs. As a solution, Chrome puts web apps and plugins in separate processes from the browser itself. Thus, a rendering engine crash in one web app will not affect the browser and other web apps, allowing the operating system (OS) to run apps in parallel to increase performance [4]. This also means the browser process will not lock up if a web app or plugin stops working. Chrome runs plugins both in-process and out of process [5].

In-Process Plugins

The WebKit's embedding layer expects an embedder to input a *WebPlugin* interface [5]. This communicates up chain to the *WebPluginDelegate* interface which talks to the NPAPI (Netscape Plugin Application Programming Interface) wrapper layer to use the plugin.

Out of Process Plugins

There is an IPC layer between the *WebPlugin* and *WebPluginDelegate* that shares code between the two [5]. There is one plugin process for each unique plugin, meaning there is one *PluginChannelHost* in the renderer for each type of plugin it uses [5]. An example of this is a situation where there are two Adobe Flash movies embedded in a web page. There would be two *WebPluginDelegateProxies* on the renderer side, which implement the *WebPluginDelegate* by sending calls over IPC to the plugin process.

Storage and Databases

The data persistence subsystem has a dependency on the Networking subsystem as Chrome stores some of the user's data in the Google cloud storage system, as well as locally on the user's machine. This is to allow for shared data between devices for things like auto-fill data, passwords and other features that increase convenience for the user. This means that the user can fill in their password once on a device and have this password stored and available on all their associated devices. Google requires a larger set of data to better tailor their ads to specific users. With the Team's proposed conceptual structure, Chrome can collect a user's search history and interests in one central location which allows for increased user understanding and an overall better ad-model performance.

Use Cases

The sequence diagram in Figure 5 describes how the Team's conceptual architecture would render the use-case where a user successfully logs into a website using a password.

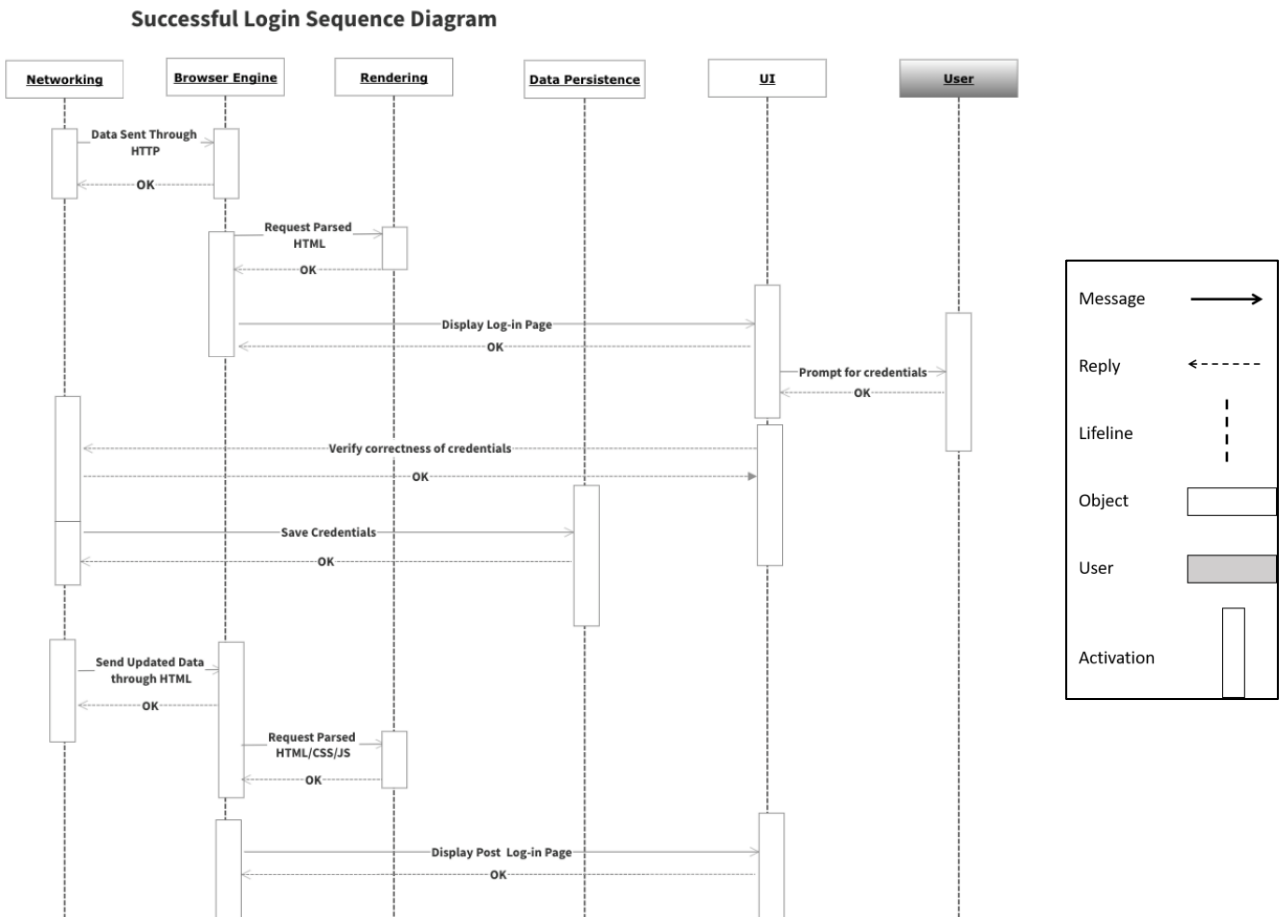

Figure 5: Successful Login Sequence Diagram

The sequence diagram in Figure 6 describes how the Team's conceptual architecture would render a page of JavaScript.

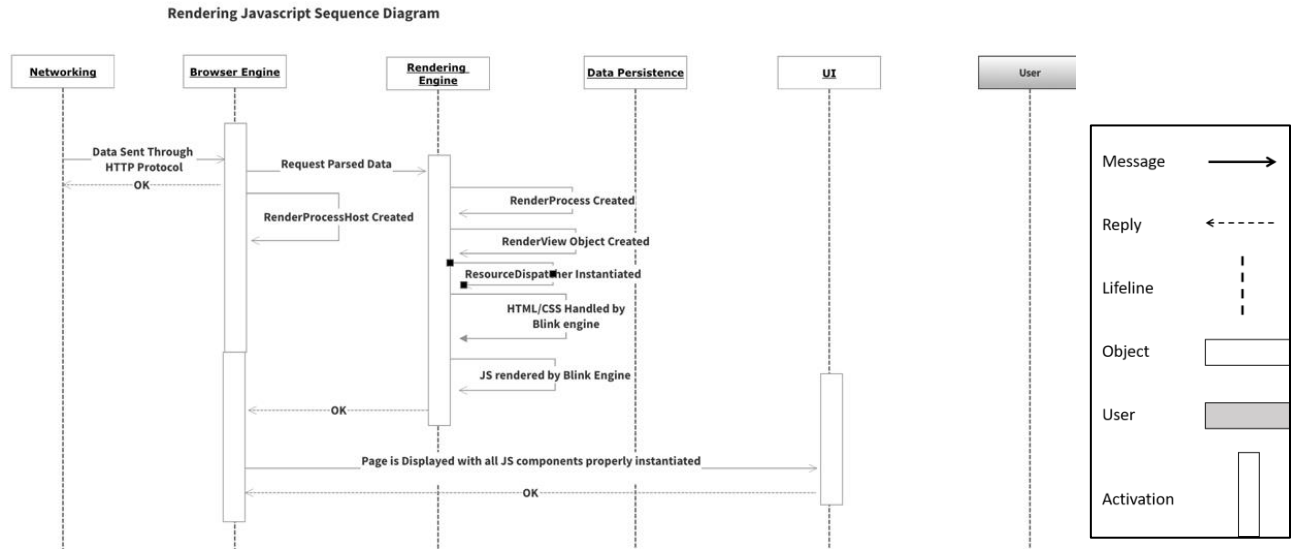

Figure 6: Rendering JavaScript Sequence Diagram

Data Dictionary

Blink: WebKit-based rendering engine

Browser: represents the browser window, contains multiple instances of WebContents

Cache: space in a computer's hard drive and in RAM memory where the browser saves copies of previously visited web pages. The browser uses the cache like a short-term memory

Channel: defines methods for communicating across pipes

Cohesion: refers to the degree to which elements inside a module belong together

Concurrency: is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other

I/O Thread: a thread in the browser process that handles IPC's and network requests, and in the renderer process that just handle IPC's

IPC Channel Proxy: Within the browser, communication with the renderers is done in a separate I/O thread. Messages to and from the views need to be proxied over to the main thread using a Channel Proxy

Main/UI Thread: a thread in the browser process that updates the UI and renderer process that runs most of Blink; responsible for rendering web pages on screen

NPAPI: Netscape Plugin Application Programming Interface, interface that allows browser extensions to be developed. The chrome plugins that have full permissions of the user and is not sandboxed

Parser: compiler or interpreter component that breaks data into smaller elements for easy translation into another language

Renderer/Render host: multi-process embedding layer

RenderProcessHost: initialized on the main thread that creates a new render process for each website

RenderView: responsible for navigational commands, receiving input events and painting web pages

RenderViewHost: receives view-specific messages from RenderProcessHost

RenderWidget: input event handling and painting

RenderWidgetHost: handles the input and painting for RenderWidget

ResourceDispatcherHost: responsible for sending network requests to the internet

Sandboxing: a security mechanism used to run an application in a restricted environment that is cut off from the rest of the computer system

Tab Helpers: separate objects that attach to WebContents

WebContents: embedded to allow multi-process rendering of HTML

WebKit: rendering engine

WebKit Glue: converts WebKit types to Chromium types.

WebKit Port: integrates with platform dependent system services such as resource loading and graphics

Naming Conventions

CSS: Cascading Style Sheets
DOM: Document Object Model
NPAPI: Netscape Plugin Application Programming Interface
I/O: Input/Output
IPC: Inter-Process Communication
NFR: Non-Functional Requirements
OO: Object Oriented
OS: Operating System
UI: User Interface

Conclusions

The team found that the object-oriented architecture helps to simplify the overall design and make development easier for a potential software team. Having each component as a separate object helps with testability and scalability. However, it is still essential for developers to pay attention to the relationships between components, as changes can end up impacting other dependent objects. Concurrency with the rendering process helps with availability, a key non-functional requirement for the system.

It is important for the team to bear in mind the performance of the system when moving forward with the design, as high performance is another important non-functional requirement of a web browser; object-oriented systems can end up being less efficient than other architecture styles, causing higher CPU and memory usage.

Due to the team's decision to omit less prioritized functionality from the architecture in interest of time and simplicity, components to handle extensions and web applications were not included in the system design. Both are central to the Chrome browser experience. Extensions and web apps are both installable components that can be created by Google or third-party developers. Extensions embed new functionality to the user interface and persist across all websites visited by a user, while web apps work as standalone, interactive features that are less monolithic than a website [6]. Moving forward, the team plans to study these two components and implement a way for them to be included within the designed architecture.

Lessons Learned

When doing research, the team quickly realized that finding a cohesive, accurate and complete explanation of the entire Chrome architecture would not be possible. Though the Chromium project has detailed documentation surrounding individual processes, much of the information is very low-level which made it difficult to create a high-level picture of the system. Though based on the same structural code, Chromium and Chrome differ slightly with their implementation and available functionality. Most of the sources used for research were based on the Chromium project and were multiple years old. This creates the possibility that some of the information studied was not an accurate representation of the current Chrome system, as

the proprietary version has strayed away from the open-source project over the past few years. For example, Chrome has deprecated its usage of plugins—except for the Flash and PDF viewer—and exclusively focuses on extensions and web apps now for its developer features, whereas Chromium still has complex architecture dedicated to running the third-party plugin processes. For the next assignment the team will be careful to consider the discrepancies between the architectures described in different sources.

The team also found it difficult to divide work evenly throughout the research and design phases of the assignment. As it was important for each team member to have a general understanding of the entire system and its interactions, tasks could not be easily divided by the separate system components. In addition, more time should have been spent planning out the conceptual design to ensure that all alternative options had been thoroughly explored and evaluated. The team rushed through the derivation process, potentially risking overlooking better architecture options. Moving forward, the team will ensure it systemically compares various ideas before making a definitive decision.

References

[1] Chromium Project, "How Chromium Displays Web Pages," 2012. [Online]. Available: https://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome. [Accessed 2018].

[2] The Chromium Project, "Inter-process Communication (IPC)," 2012. [Online]. Available: https://www.chromium.org/developers/design-documents/inter-process-communication. [Accessed 2018].

[3] The Chromium Project, "Multi-process Architecture," 2008. [Online]. Available: https://www.chromium.org/developers/design-documents/multi-process-architecture. [Accessed 2018].

[4] Chromium Blog, "Multi-process Architecture," 2008. [Online]. Available: https://blog.chromium.org/2008/09/multi-process-architecture.html. [Accessed 2018].

[5] The Chromium Project, "Plugin Architecture," 2010. [Online]. Available: https://www.chromium.org/developers/design-documents/plugin-architecture. [Accessed 2018].

[6] M. Mahemoff, "Extensions and Apps in the Chrome Web Store," Google, 2010. [Online]. Available: https://developer.chrome.com/webstore/apps_vs_extensions. [Accessed 2018].