Chrome's Conceptual Architecture - Derived by uBlock Origin

Nicolas Merz, Zach Yale, Heather Geiger, Darrien Park, Kurt Blair, Dan Kailly

October 19, 2018

Abstract

Software functions often span multiple conceptual modules making the extraction of conceptual architectures difficult. In this report, we derive the conceptual architecture for the Google Chrome web browser to illustrate an understanding of the architecture of a system. Since there is no high-level, recently published, conceptual for Chrome, we had to manually derive a representation from available documentation and other external research. We propose a combination of layered, object-oriented, and implicit invocation architectural styles and explain the key subsystems involved. Our chosen subsystems are the UI, Browser Engine, Data Persistence, Renderer, Network, and Plugins. We provide an in-depth breakdown of the UI and the Plugin sub-subsystems to demonstrate dependencies within the module and interactions with other subsystems in the overall architecture. Furthermore, we touch on how three external interfaces (Cache, Graphics Rendering, and Operating System) interact with the Chrome browser. To illustrate our derived subsystem dependencies, we present a Sequence Diagram for two Use Cases in the browser; we show how a user logs into a webpage and Chrome saves the password, and we show how Chrome renders a page with JavaScript. In sections 6 and 7 we discuss the role processes and threads for concurrency in Chrome and the browser's evolution. Finally we discuss our research on Team Issues in Chrome Development and touch on the limitations and lessons we learned while deriving Chrome's conceptual architecture.

Contents

1 Introduction

The Chrome browser was released by Google in September 2008. Powered by the open-source Chromium Project, the multi-platform browser is maintained by both Google employees and volunteer open-source contributors. For this course project assignment, the team has been tasked with deriving a conceptual architecture for Chrome. The conclusions drawn in this report have been derived from existing Chrome and Chromium documentation, developer discussions, and some assumptions from the team. Although there is extensive documentation for Chrome, we did not find a overall architectural representation. Examining this documentation, we derive a conceptual architecture for Chrome.

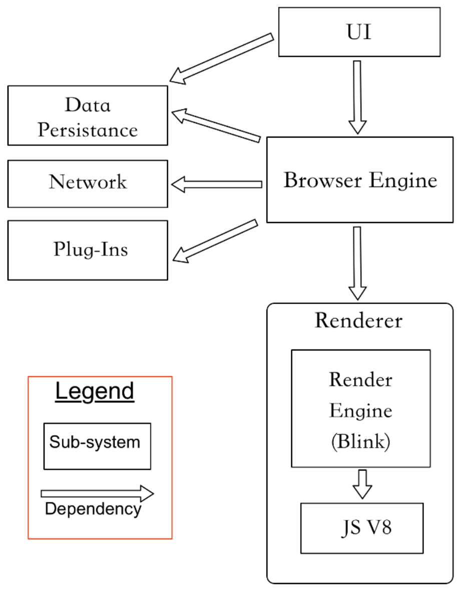
This report presents our team's derivation of a conceptual architecture for Google Chrome, based off of researched documentation and developer experiences working on the Chromium Project. We explain how our team developed the architecture structure, and will explore the logic behind the architecture derivation process. It will also examine the individual components and interactions between the included architecture subsystems, while documenting the system evolution history of Chrome. We have concluded that Chrome's architecture is a combination of layered, object-oriented, and implicit invocation styles. This derived architecture will be examined by evaluating data flow among parts, two use cases, and a subsystem breakdown of two components. Concurrency in Chrome will be evaluated, and external interfaces for the architecture will be reviewed. Finally, team issues within the Chromium Project will be discussed, as well as limitations and lessons learned within our team.

2 Conceptual Architecture

2.1 Derivation Process

Our derivation process for Chrome's conceptual architecture first began in CISC 326/CMPE 322 lectures where we learned about the Reference Architecture for Web Browsers (Hassan). Each group member then began independently researching Chrome and Chromium-specific documentation. Everyone then drafted their own version of the conceptual architecture based on their understanding of the information they had read. We then met as a group to compare and discuss how in-depth we should get, which modules to include and exclude, and how to structure the diagram spatially. Once we agreed on a version, we met with our TA for further feedback. We learned that a two-way dependency between Renderer and Browser Engine, for example, would take away from our claim that Chromes architecture was layered, so we iterated over the diagram again using the TA's provided feedback. Figure 1, is the diagram we decided best represented the Conceptual Architecture of Google Chrome.

2.2 Proposed Conceptual Architecture



We propose the architecture to be a combination of Layered, Object-Oriented, and Implicit Invocation styles. We think it is layered because of the strict layering between the UI, Browser and Renderer. In the layered style, each layer provides a service to the layer above and acts as a client to the layer below (Garland and Shaw). The Browser provides a service to the UI by calling the Renderer to obtain HTML, CSS and JavaScript content to display, it calls the Network to get a network connection to load the URL, calls any applicable plugins and calls data persistence to obtain any user saved information and preferences. The Renderer provides the service of interpreting and parsing HTML and XML code, formatting according to CSS and XLS specifications, and calls the JavaScript V8 engine to load any applicable JavaScript. The advantages of this are the increasing levels of abstraction which helps to partition large problems, such as handling many site instances at once, and the support for enhancement and reuse (Garland and Shaw). Reuse is important in Chrome because the code is constantly being changed and updated.

We also propose the architecture is object-oriented because modules are encapsulated and invisible to other modules. Modules can make explicit calls to others. For example, the Browser Engine calls the Network module to access a web page's server to obtain confirmation of a user's login credentials.

Finally, we think Chrome also uses Implicit Invocation in at least one situation. In Chrome, there are two primary types of messages sent from the Browser to the Renderer: “routed” and “control.” Control messages are sent from the Browser and do not specify a certain rendered process (not frame specific). Routed messages specify a process'unique ID. Typically just this one specific render process receives the routed messages. However, any class/process can receive routed messages by registering itself with the route using listeners (Inter-Process Communication - Chromium Design Documents). This newly registered process now receives the ”routed” message that via their unique ID. A process registering itself with the message is advantageous for on the fly configuration and reconfiguration of web pages. The Browser Engine acts as the broadcasting system and invokes all the processes that have registered themselves with the “event, or message in this case (Garland and Shaw).

2.2.1 UI

The User Interface is the front end of the Google Chrome browser. It displays rendered HTML with CSS and any applicable JavaScript and Graphics content. The UI also provides access to inspect the HTML and JavaScript elements (Rahman, Rigby, Shihab) in the browsing window. The UI's dependency on Data Persistence enables the display of user browsing history, user textquotesingle s saved preferences, and the auto-fill of previously login credentials. Within the UI module is the UI backend that controls the makeup of the UI module, including tabs, the toolbar, and the bookmarks bar. It represents the views in the DOM tree. The backend depends on the Browser Engine to collect rendered content from the Renderer through the Browser Engine (Rahman, Rigby, Shihab).

2.2.2 Browser Engine

The Browser Engine is the core of the Chrome browser. It connects to UI to the Renderer by acting as a communicator between the two. The Browser Engine initiates all web page events like loading a URL, form submission, re-loading and navigating (Rahman, Rigby, Shihab). Extensions: We decided to not to get too granular in our conceptual architecture so we grouped extensions within the Browser Engine.

- **Extensions:** Extensions allow users to make use of third party features to enhance their browser. Extensions are controlled by the Browser Engine, so although not pictured as a separate entity, we want to mention their inclusion here.

2.2.3 Data Persistence

Data persistence is the module to store data, like passwords, browsing history, and user preferences (Rahman, Rigby, Shihab). Passwords are saved on the user's machine for enhanced security. This module helps to create a more seamless browser experience by auto-filling login credentials, saving browsing history (while not in incognito mode), and loading user preferences so they can be quickly accessed.

2.2.4 Renderer

The Renderer is the module that parses information contained in the URL. We broke the Renderer into 2 key sub-subsystems, although we acknowledge that there exists more sub-subsystems in the Renderer, we chose to display the two largest, most prominently used ones.

- **Render Engine:** Render Engine (Blink): Renders a web page from a URL. It acts as an interpreter for HTML and XML and formats according to CSS and XSL (Rahman, Rigby, Shihab).

- **JavaScript V8:** The Render Engine depends on the JavaScript V8 Engine to process JS contents. Blink also is responsible for creating the DOM tree through multiple iterations to prepare it for rendering. JavaScript V8 Engine: Processes JavaScript content for the page, where required.

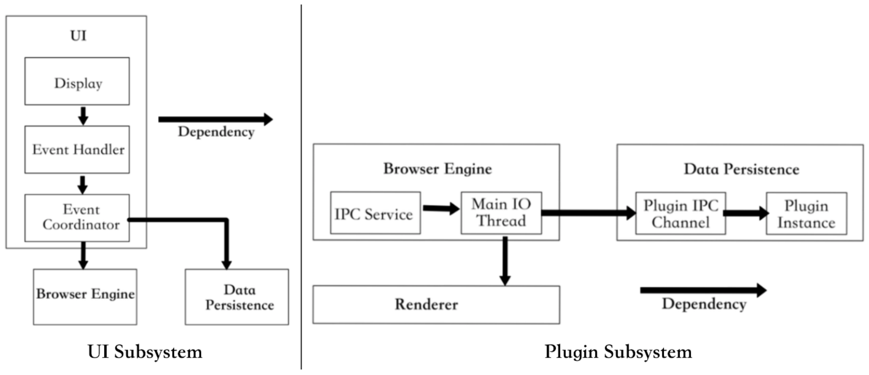
2.2.5 Network

The Network module interfaces with the hardware to allow network connections and connections to other third party tools hosted on the network (Rahman, Rigby, Shihab). A network connection is required to access a web page's server. The Browser depends on the Network to find an internet connection in order to load a web page. The Browser Engine also calls the network to validate a user's login credentials, for example.

2.2.6 Plugins

Plugins are apps hosted on the user's machine like Adobe Flash, for example. The Browser depends on plugins as it will call it when a site instance requires access to a plugin. A single process instance is creating for each unique plugin connected to the Chrome browser.. This means that since there is only 1 process for Flash, if Flash crashes, then all web pages using Flash will crash too (Plugin Architecture - Chromium Design Documents).

3 Sub-system Breakdowns



3.1 UI

The UI module interacts depends on the Browser Engine and Data Persistence. The UI's interactions with Data Persistence allow for saving browsing history. The UI also interacts with the Browser Engine which connects to the Network module and Renderer to render a web page. When a user chooses to have the password saved, the Browser Engine connects to Data Persistence to save this information locally on the user's machine.

The sub-subsystem architecture for the UI is layered with Display at the top with direct interaction from the user. Once the user interacts with the Display, by either clicking a web page or requesting browser history, the user's action is sent to the Event Handler. The Event Handler calls the Event Coordinator to determine which external component is involved with the action. The Event Coordinator sends data to the Browser Engine, the core of Chromearchitecture. When requesting a user's browsing history, the Event Coordinator will interact directly with Data Persistence.

3.2 Plug-Ins

Plugins are a distinct process that is separated from the Browser Engine. One of the justifications behind this high-level design decision is that when a distinct plugin crashes, the separation of the plugin process from the Browser Engine results in Chrome continuing to operate normally despite the plugin failure.

Chrome plugins can be run either in-process, or out-of-process. An out-of-process plugin can be run outside of the Browser Engine and Renderer, and communicates with the Browser Engine using an IPC pipe allocated for the specific plugin process - an example being an Adobe Flash movie embedded in a web page. Each end of the IPC channel maps to different instances of a plugin, and is capable of multiplexing communication between these instances over a single IPC pipe.

An in-process plugin begins in an embedded layer that communicates with Blink. The plugin communicates directly with the Renderer through an IPC pipe, and uses the WebPluginPageDelegate interface to create a unique sandboxed renderer view for the plugin. This leads to inconsistencies with our conceptual architecture due to the lack of dependency between Plugins and the Renderer. To be consistent, an in-process plugin resides in the browser process, which then communicates with the appropriate rendered processes using allocated IPC pipes and a master I/O thread in the Browser Engine to relay critical messages. Attached below is a diagram of an out-of-process plugin.

4 External Interfaces

- **Cache:** Website information can be stored locally in a cache to reduce the amount of information that needs to be fetched remotely. The external interfaces used for this are the Backend (used for enumeration of resources) and Entry (used for operations for a given resource) interfaces. All cache information is stored in a folder called cache, and this folder contains an index file and four data files. The index file contains the hash table which allows Chrome to find the required data stored in those four data files. Chrome usually uses the memory cache first due to its high-speed and responsiveness, but will also store web page information into non-volatile storage (ie. the disk cache) in case of crashes or shutdowns.

- **Graphics Rendering:** Pages can be rendered using a GPU if it is available. The compositor (a part of the Renderer) can interface with Chrome's GPU process to render pages. The compositor places its processed graphics into some shared memory (such as a bitmap), which can then be accessed by the GPU process and be supplied to the required graphics calls.

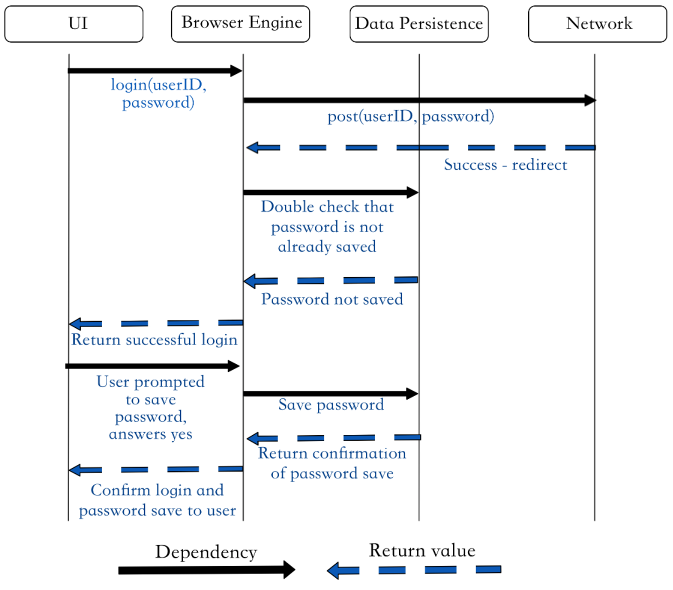
- **Operating System:** Features such as sandboxing and plugins leverage resources provided by the operating system to perform their required functionality. The architecture of sandboxing and the assurances that it provides is specific to each OS, however it utilizes the on-board security protocols to allow code execution that cannot permanently make changes to the computer or access confidential information. Plugins, on the other hand, utilize packages and frameworks installed on the OS to enhance or expand the capability of the browser. These resources need to be OS-bound because they are usually developed by third-parties and are often the cause of browser instability.

5 Use Cases

5.1 User Login, Chrome saves password

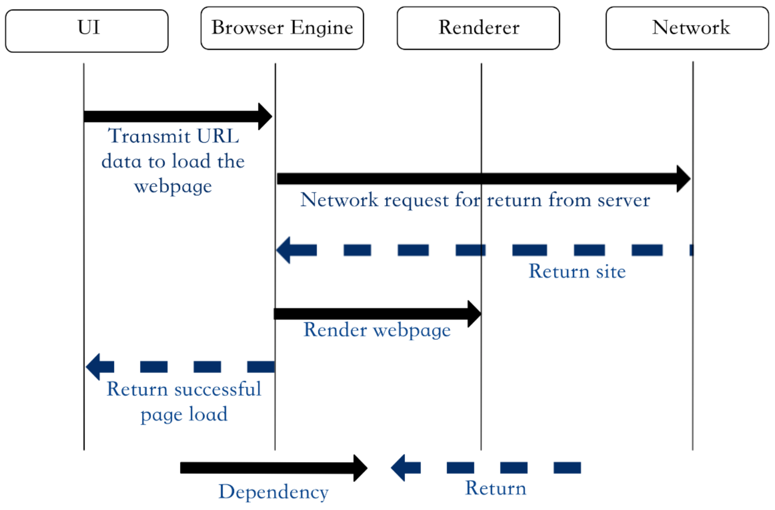
For the first use case, we evaluated the scenario of a user successfully logging into a website, followed by Chrome saving the password. The scope of this use case only covers the act of logging in and saving the password, and not the rendering of a page as that use case is covered in Use Case 2. We also make the assumption that the user has not previously saved their password on this website. The sequence starts as the user enters their credentials into the UI which sends login form data to the Browser Engine. The Browser Engine then holds a copy of the credentials, and posts the login attempt to the Network. The Network communicates with the website's servers, and upon a successful login, the server redirects the user to a new web page. This is done by returning a redirect from the Network to the Browser Engine. Upon the successful login redirect, the Browser Engine checks that the website's login credentials have been previously saved, by sending the website credentials to Data Persistence to check for duplicates or changes. Browser Engine then calls UI with a request to prompt the user whether or not they want to save their password. If the user clicks save, the UI returns the save request to the Browser Engine.The Browser Engine then sends the credentials to Data Persistence. Data Persistence sends a successful return value to the Browser Engine which confirms with the UI that the credentials have been saved.

Please note that two changes exist between this report copy of Use Case 1, and the version presented in our presentation. In the presentation, the Browser Engine prompted the UI to ask the user whether or not they wanted to save and added a double check to ensure no credentials had previously been saved on the site.



5.2 Render a page with JavaScript

For the second use case, the scenario being evaluated is Chrome rendering a page that has JavaScript. This use case begins with the user entering a URL to the UI. The UI sends a request to the Browser Engine to load the page using the URL. Browser Engine sends a request to the website's server through the Network module to load that web page. The Network returns the web page to the Browser Engine, which then requests the Renderer to display the website. The Renderer returns to the Browser Engine, indicating successful interpretation of the web page, and the Browser Engine returns to the UI that the page has been successfully loaded.


UI
Browser Engine
Renderer
Network
Transmit URL data to load the webpage
Network request for return from server
Return site
Render webpage
Return successful page load
Dependency
Return

6 Concurrency

Google Chrome is one of the earliest browsers to benefit from concurrency. Traditionally, browsers were single process, but Chrome pioneered the splitting of a browser into multiple modules. Each browsing process is split into seperate, sandboxed instances. These processes have the ability to communicate with the Browser Engine through Inter-Process Communication (IPC).

One of the most important uses of concurrency is the one-to-many Browser Engine to Renderer relationship. Chrome has become renowned for the security and stability in this concurrency model. The Browser Engine is responsible for spawning instances of Renderer when new tabs are created. The Browser Engine acts as a controller for these Renderers when switching between tabs in a given process, and as a bridge to the Network module when routing a URL through the Browser Engine to the Network stack. The RenderProcess component of the Renderer is responsible for all message passing between its instance of RenderProcessHost in the Browser Engine. The contents of a web page are communicated through the RenderView and RenderViewHost components located in the Renderer and Browser Engine, respectively (Multi-process Architecture - Design Documents).

Site Isolation is the default model to group tabs. This is a recent development to Chrome (made the default model in Chrome 67.0), which replaces the process-per-site-instance model. Site Isolation strictly enforces that only tabs on the same site can be placed within the same process. However, in Site Instancing, two tabs on the same site can be in different processes, as long as tabs that script each other are located in the same process. Site Isolation differs from process-per-site-instance in that the latter allowed for multiple sites to be in the same process, such as in Renderer-initiated navigation (i.e. clicking a link on one site that leads to another site). To reduce memory overhead, if Chrome had too many open processes, the process-per-site instance model would reuse a process even if the sites in that process were not connected.

The idea behind this model is twofold; it improves stability and security. A tab crashing on a site will not affect other tabs. For instance, if a user is watching a video and is reading a news article, the crashing of the video player will not affect the site hosting the news article, and will keep the browser responsive. Since these tabs are isolated from each other, their communication is limited and controlled, enhancing security. Security exploits such as Cross-Site Scripting used to be more common, since cookies from one site could be stolen by another when open in the same browser. However, Chrome's sandboxing now prevents this kind of exploit code from working.

Within these processes, multi-threading also helps to achieve concurrency. Threads are used to separate different responsibilities; the main thread types are IPC and processing. All processes have an I/O thread, used for IPC, and a main thread. The benefit of multi-threading is that a process can remain responsive to messages and add them to the process queue while processing concurrently. For example, having an I/O thread allows the UI to remain responsive without being slowed by expensive computation.

Generally, processes also have some general purpose and specialized threads. Blink, for example, has a main thread, some internal threads, and an arbitrary number of worker threads. Its main processing of interpreting JavaScript and XML are done on the main thread, while the worker threads can handle services such as Worklets or ServiceWorkers (How Blink works - Chromium Design Documents).

Finally, Chrome developers are encouraged to use sequencing instead of asynchronous execution as these are run in posting order on whatever thread is available (Threading and Tasks in Chrome). This improves thread-safety without using locking mechanisms, which could hog important resources.

While the Site Isolation model has many benefits, it suffers from expensive memory overhead. When the default concurrency model was changed in Chrome 67.0, memory usage increased by 10-13 %. Implementing this model is more difficult than a single process or process-per-tab model, as complex logic is needed to switch tabs between sites and for inter-site communication.

7 Chrome Evolution

Chrome has gone through many iterations since its first beta release in 2008; moving from an unstable, barebones browser to an incredibly robust one, and becoming the most popular browser in the world. Chrome's first official, stable cross-platform release (Chrome 5.0) was in May of 2010. Since then, the Chrome team has taken user feedback to implement new features and perfect existing ones, improving the usability and performance of the system. The incredibly popular Extensions feature did not exist before the release of Chrome 5.0. The earlier stages of Chrome's development also saw the release of form auto-filling, the Chrome Web Store, and password syncing. Some large changes later on include encryption for all syncing in 2011, the release of Android and iOS clients in 2012, and the change from WebKit to Blink in 2013.

As more resources pour into Chrome, the number of files has increased drastically, representing its increased rate of development. This can be seen in Table 1 by the growing number of major releases year over year.

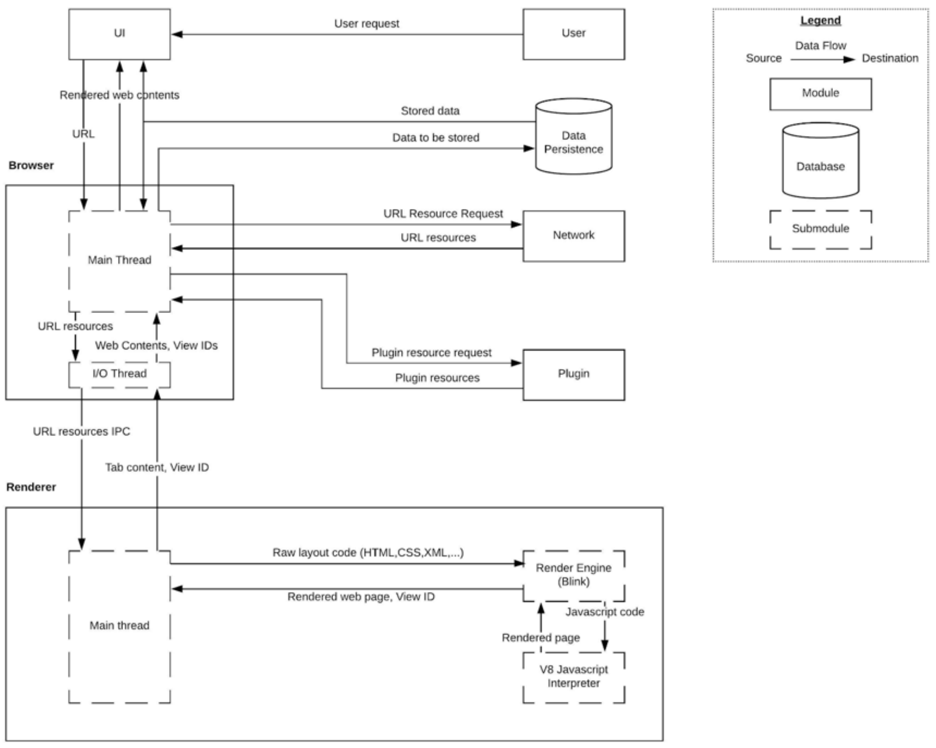
Year	**Chrome Release**	**Number of Files**
2010	5.0	7200
2011	13.0	10200
2012	22.0	17200
2014	34.0	28500

Table 1: Evolution of Chrome

Chrome has seen large changes in terms of modularity and cohesion. In earlier releases, many features spanned multiple modules, which lead to high coupling and low cohesion (some spanned more than 10 modules). However, in release 34.0, Chrome developers attempted to remove all dead code. This reduced how many modules certain features touched. In the dead code, there were a lot of unused feature toggles which were not even fully implemented, but since their code was in the codebase they created artificial dependencies between modules. In release of Chrome 34.0, features now span between 2 to 5 modules, which allows module changes to be less complicated (Rahman, 2018).

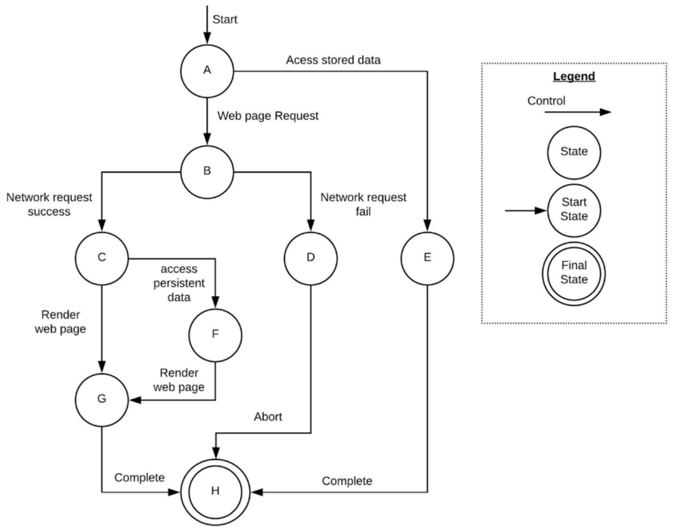
8 Data and Control Flow

8.1 Data Flow



Data flow begins in the UI when a user initiates an event on a web page. When the request to access a URL is made, the UI calls the Browser Engine and data flows from the UI through Browser Engine to Renderer. When a web page request is made, the browser calls the network stack to obtain page information to and calls any third-party plugins installed on the OS needed by the page. Information from the network stack is received by the main thread of the Browser Engine and routed to Renderer where the page and form layout can be interpreted. Blink interprets the layout while the JavaScript V8 Engine executes JavaScript code and obtains the rendered elements. The fully rendered page is assigned a unique view ID by Renderer and is sent back to the Browser Engine, where all rendered pages are collected and associated with their corresponding tabs. The rendered pages displayed by the UI, where page elements wait for the next user input. Data flow to Data Persistence begins when the user saves browsing data. The data to be saved flows from the UI to the Browser Engine then to the Data Persistence.

8.2 Control Flow



Data flow is accompanied by Control flow. Data flow shows the movement of data between different entities in the system while control flow defines the workflow of tasks executed in order. It shows where control starts and ends, and where it may branch in certain situations. From the start state, Chrome allows the user to request access to a web page, or to access local storage to display browser history, passwords, logs, etc. When a request to a web page is called, the system communicates with the network stack to obtain the required resources if a successful internet connection is detected. However, if these resources are stored in either the memory or disk cache, then there is an option to branch and the resources will be retrieved locally. Once the data is retrieved, the page may be rendered and the control ends. If any failure is encountered in the network stack, the browser will abort and proceed to the finish state.

9 Chrome Team Issues

When looking at the Chromium team documentation, there is a total of ten sub-teams assigned to different components of the browser. This should warrant further scrutiny as documentation is highly outdated. Many of the documents have not been updated since 2013, with some diagrams and documentation pages reaching nearly a decade since being uploaded to Chromium documentation. The Chromium documentation states that there are more than 20 top-level projects, and each sub-team has roughly 5 members. This raises questions on how the development team is organized. A reasonable conclusion as to why this is can be found in a Medium article written by Aaron Boodman, a former Chrome engineer who was responsible for implementing sandboxed security features currently found in Chrome's extensions.

What is most interesting about Chrome's development is that for a project of this scale and complexity, it is unusual that there is no branching made to work on major features of the browser. With the exception of maintenance branches created and terminated between consecutive beta releases, development happens entirely on the trunk or master branch of the project. The rationale behind this is based off of how expensive testing can be for the development effort. By the time a Chrome engineer merges their changes into trunk, the current trunk is be so vastly different from their merged branch, that integration testing would become difficult and very lengthy to perform. Of course, there are positive trade-offs for this development practice too. All Chrome developers are now on the same page, and are always working on the most current version of the project. Chrome engineers are also allowed to make changes to modules outside of their team's responsibilities. All an engineer must do is ask the current sub-team tasked to the module for permission to edit. This direct approach to refactoring the Chrome codebase stems from having extensive coverage from automated testing (over 12,000 class-level tests and 2,000 automated integration tests for Chrome exclusively, not including Blink and V8). Such an approach means that one group's area of development is likely to be changed by a different sub-team's commit. Boodman notes that constantly checking to see if your team's component is working correctly can lead to a “very real issue of stress during development.

10 Our Limitations and Learnings

Throughout the derivation process, we encountered challenges that affected the decisions towards the creation of our final conceptual architecture. A considerable amount of our research was obtained from the Chrome/Chromium documentation which is drastically out of date. Chrome was first made available to the public in 2008, and the last known updates to the documentation were pushed in 2014. Based on the release history, Chrome is updated approximately every 2 months and therefore the integrity of the information we found is questionable. This raises the issue of inconsistent information across our sources and affects the functionality and relevance of Chrome's subsystems interpreted by our team. There was also a lack of firsthand information available to us. Very few Chrome developers have published their development process publicly, so much of our research comes from interpretations of the system. The requirements to make contributions to the Chromium project stringent.

We have some valuable takeaways from this assignment that would definitely be beneficial for future assignments and projects. We have learned that a large majority of time was required to work through the vast amounts of documentation and derive the high-level architecture. Since chrome is a very complex program, the documentation needs to be carefully understood so as to avoid ambiguity and it can be difficult to synthesize such a large amount of information into a 15 page document. Furthermore, we learned to minimize coupling and increase cohesion by reducing the number of modules in the architecture despite the challenge of breaking down such a complex system and as a team we developed better communication, time management, and group organization skills.

11 Conclusion

We concluded that Google Chrome is a combination of layered, object-oriented, and implicit invocation architecture styles. The external interfaces with which Chrome interacts and relies upon includes the cache, the graphics processing unit, and the operating system. The cache allows web information to be stored locally on hardware and reduces overhead when retrieving web assets, the OS provides resources to the browser to enhance security and add extra functionality, and an external GPU device allows web pages to be rendered concurrently via shared memory from the compositor. Concurrency is also managed through site isolation and multi-threading to take advantage of the Chrome's multi-process features and the ability to asynchronously execute tasks for a responsive and efficient system.

Although we encountered limitations completing this assignment, the team managed to take away valuable skills regarding deriving a conceptual architecture, time management, communication, and organization. This study of the conceptual architecture was strictly at a high level and a deeper understanding of the software will come as we delve further into the source code.

References

[1] A. Neupan. (2013) The evolution of google chrome. [Online]. Available: http://www.revolvingweb.com/the-evolution-of-google-chrome/

[2] T. Rahman, P. C. Rigby, and E. Shihab. (2018) The modular and feature toggle architectures of google chrome. [Online]. Available: http://users.encs.concordia.ca/ pcr/paper/Rahman2018EMSE-preproduction.pdf

[3] na. (2009-present) The chromium projects - design documents. [Online]. Available: https://www.chromium.org/developers/design-documents

[4] ——. (2009-present) Threading and tasks in chrome. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/lkgr/docs/threading$_and_tasks.md$

[5] ——. (2009-present) Sandbox design in chrome. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.mdOverview

[6] A. E. Hassan. (2018) Module 5: Reference architecture (web servers and web browsers). [Online]. Available: http://cs.queensu.ca/ ahmed/home/teaching/CISC322/F18/slides/CISC322$_05_ReferenceArchitecture.pdf$

[7] D. Garlan and M. Shaw. (1994) An introduction to software architecture. [Online]. Available: http://cs.queensu.ca/ ahmed/home/teaching/CISC322/F18/files/Introduction$_SoftwareArchitecture.pdf$

[8] na. (2009-present) The chromium projects - plugin architecture. [Online]. Available: https://www.chromium.org/developers/design-documents/plugin-architecture

[9] A. Boodman. (2015) How chromium works. [Online]. Available: https://medium.com/@aboodman/in-march-2011-i-drafted-an-article-explaining-how-the-team-responsible-for-google-chrome-ships-c479ba623a1b

[10] na. (2009-present) The chromium projects - teams. [Online]. Available: https://www.chromium.org/teams/

[11] ——. (2009-present) The chromium projects - getting around the chromium source code directory structure. [Online]. Available: https://www.chromium.org/developers/how-tos/getting-around-the-chrome-source-code

[12] P. Barth, S. Felt, and A. Boodman. (2009) Protecting browsers from extension vulnerabilities. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-185.pdf