October 19th, 2018

Chrome Conceptual Architecture Report

Authors
Brynnon Picard (15bdrp@queensu.ca - #20005203)
Roy Griffiths (18rahg@queensu.ca - #20137434)
Alex Galbraith (18asrg@queensu.ca - #20135646)
Sam Song (15shs1@queensu.ca - #10211857)
Bradley Kurtz (15bdk2@queensu.ca - #20020794)
Dongho Han (16dhh@queensu.ca - #20027554)

1.0 Abstract

This report describes our conceptual architecture of Chrome, outlines our derivation process, and explains our rationale for each subsystem, module and connection in our conceptual architecture. Chrome is a complicated, professional piece of software, and as such provides an excellent case study for those interested in Software Architecture. The architecture of Chrome is driven by its functional and non-functional requirements (NFRs), and the team of programmers, designers and architects behind the browser. Over the years, Chrome has changed dramatically with entire modules of code being replaced with new proprietary software, and the fundamental design goals shifting from speed and stability to speed, security, stability, and reliability [1].

In order to derive an accurate and up-to-date conceptual architecture, we have taken into account all the aforementioned factors. We used documentation from the open source analogue, Chromium, to identify major subsystems and crucial design elements such as Chrome's multi-process architecture, use of implicit invocation, and Renderer/Browser decoupling. Running through use cases allowed us to identify the less important but nonetheless crucial subsystems, Network, Graphics, Extensions, and Plugins. The non-functional requirements Google enforced upon Chrome, especially security and stability, guided us when we began to connect these modules up. Google specifically wanted to ensure stability between multiple tabs, and to sandbox the rendering (this includes running scripts) of each individual page. After laying out our components, we ran through use cases and generated possible sequence diagrams. The diagrams that agree most with Google's non-functional requirements were then used to link up our subsystems.

Ultimately we landed on an Object-oriented architecture with implicit invocation for communicating between the Browser and Renderer. Chrome also has a unique multi-process architecture where each rendering instance runs in a separate process. This helps enforce stability and security between Renderers. The Browser subsystem acts as a central control and manages communication between all other subsystems while running the UI and handling data persistence. Connected to the Browser are Network, Graphics, Plugin and Extension subsystems which the Renderers may access via the Browser subsystem.

2.0 Introduction

The purpose of this report is to explore the conceptual architecture of Google Chrome and explain the process our team used to derive our final conceptual architecture. The report has three main content sections. The first section is derivation where we will discuss how we derived our architecture and the rationale for many of the major decisions. The second section describes our final architecture in detail, including a description of each subsystem and module, as well as the connections between them. The third section discusses limitations in our analysis and lessons we learned during the process of derivation. Finally, we will wrap everything up with a conclusion, followed by a list of references.

Our derivation of Chrome's architecture is driven by both the functional and non-functional requirements of Chrome. Functionally, Chrome can be separated into two groups. The general web browser functionality of displaying interactable pages, and the Chrome functionality including UI, user data management and tab support. We used these major functionalities to roughly block out the subsystems we need. The non-functional requirements then impose a more rigid structure on these subsystems. The concurrency, security, and stability NFRs specified by Google force the functions of Chrome to be grouped together in a very specific manner, and communicate with each other very selectively.

Chrome was built on four core principles: speed, simplicity, stability, and security. Development of Chrome first started with speed as its main priority; Google wanted to make the fastest browser possible. This led to the creation of Blink and the V8 Engine which is considered to be one of the fastest JavaScript engines [11]. As the browser landscape evolved Google focused on security and stability. Chrome first implemented the principle of least privilege and process isolation. The principle of least privilege allows Chrome code to be sandboxed and its privileges to interact with the host system to be restricted. Untrusted (web) content is passed off to sandboxed code for consumption. On the other hand, for stability process isolation is extensively used. All processes in the system are isolated from each other, even the plugins. This allows maximum stability to the browser, one compromise in a tab will not lead to a security risk in other tabs and single failure in a tab will not cause the whole system to go down.

In order to achieve these feats of stability, security and concurrency, Chrome has two main subsystem, the Browser and Renderer. Multiple Renderer instances may exist at any given time, oftentimes with one Renderer handling only a single tab. Ideally, Renderer instances are isolated from the OS, able to communicate solely with the Browser via inter-process communication (IPC). This communication relies on Chrome's underlying implicit invocation architecture with IPC messages triggering implicitly defined functions on both the Renderer and Browser side [3]. The Renderers are fed information from the Network and Extension subsystems via the Browser, and may access OS resources which are abstracted in the Browser, Graphics, and Network subsystems, all via the Browser. Plugins, which are being phased out by Google [9], may also be accessed by Renderers (ideally via the Browser) for imbedding small programs like Flash.

3.0 Derivation

Our derivation process was driven primarily by the functional and non-functional requirements of Chrome. Our derivation occurred in two main stages. First, we identified the primary functional requirements of Chrome. From this, we built an initial conceptual architecture. Next, we looked into the non-functional requirements of Chrome, specifically concurrency, stability, and security. These NFRs drastically changed our architecture and gave us our final conceptual architecture.

3.1 Functionality and Use Case Diagram

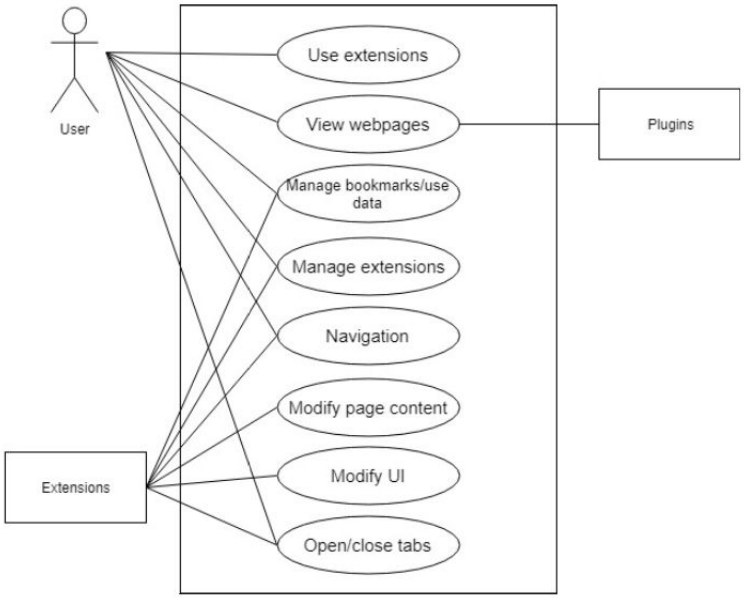


Figure 1: Use case Diagram

We divided the functionality of Chrome is divided into two parts (See Figure 1): web browser and Chrome functionality. Web browser part of functionality describes all operation web browser is built for, of which we identified 5 main functions: Rendering of HTML and CSS, running Javascript, rendering 2D and 3D graphics, plugin integration, and accessing this information from the internet. Meanwhile, Chrome functionality illustrates functions that Google implemented but are not exclusive to Chrome. Chrome functionality includes user data management (bookmarks, history, etc), user data synchronization via Google's cloud, extension support, plugin management, and multi-tab support. Once we had these, the issue became grouping these functions into subsystems and organizing the relationships between them.

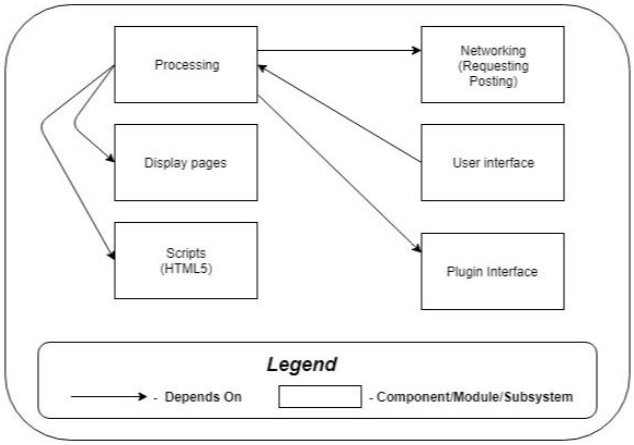


Figure 2: Initial conceptual architecture

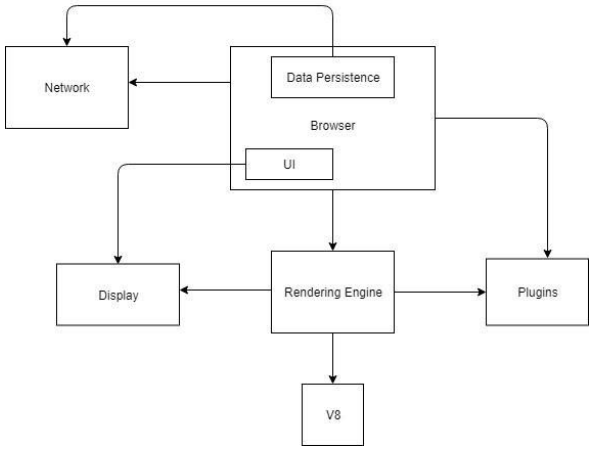


Figure 2.1: Updated Concept

In our first draft, we mistakenly grouped plugins and extensions together, forgot about graphics altogether, and separated our rendering from our scripts (See Figure 2).

We identified and remedied some of these issues and landed upon the architecture seen in Figure 2.1. Processing was renamed to Browser, UI became included in Browser. “Display pages” became Rendering Engine and Scripts were changed to V8. At this point we were still missing Extensions, and had V8 outside our Rendering Engine. Next, we looked into the non-functional requirements of Chrome, specifically its concurrency, stability and security requirements.

3.1 Concurrency, Stability, and Security

One of the major driving factors in the design of Chrome’s architecture was concurrency. The Google team wanted a browser that ran multiple tabs both securely and in a stable manner. At the time of Chrome’s conception, most browsers would fully lock up in the event that even a single tab crashed. Google had a solution: Chrome would run multiple processes meaning that if one tab crashes, the rest of the browser would be unaffected. Google decoupled the rendering subsystem from the browser subsystem, thus allowing pages to be rendered independently of each other [2].

Not only does separating the renderers into separate processes prevent crashes from affecting the whole system, but it also prevents a compromised renderer from accessing data on other rendering processes. To put this in more concrete terms, if tab A and tab B are running in different processes we get the following side effects:

- If tab A crashes, tab B is unaffected and can still be used as normal.
- If tab A is compromised by malicious javascript, it cannot access data on tab B. For example if on tab A you get some dodgy popup, and tab B is running Facebook, the dodgy popup can't scrape your Facebook feed for personal information because they cannot communicate.

With each renderer now running its own process, the core browser module was left in its own process. The browser’s job then became managing the rendering processes. To communicate with rendering processes, Chrome uses a message based form of interprocess communication [3]. Here, Google opted for implicit invocation to be implemented to handle IPC messaging; When the browser or renderer receives an IPC message, the respective module invokes an implicitly defined function to handle the request. So now we know two things:

1. Chrome has at least two main modules, the Browser, and the Renderer.
2. Chrome uses an implicit invocation architecture for interprocess communication.

Now that the browser and rendering are decoupled, Chrome is able to enforce a wide range of security features on the rendering instances. Google decided to use the Browser subsystem (Figure 3) to sandbox and abstract the rendering processes from the OS. The renderers are given access to the network and file system of the operating system only through communication with the Browser subsystem. Ideally Graphics too would be sandboxed in this manner, so we have modelled our architecture as such (Figure 3), but in reality, this would likely cause performance issues. This gives us our third insight into the Chrome architecture:

3. The renderer module is connected to all other modules only through access to the browser.

Finally, there is one more insight to be gleaned from Chrome's concurrency. On the browser side, there are two threads, the IO thread which deals with IPC, and the UI thread, that runs the UI. [4] This gives us two of the modules we selected for our browser subsystem: the kernel which deals with IO and decision making and the UI which runs the user interface. Applying what we now know about Chrome's architecture, we settled upon our final conceptual architecture.

4.0 Final Architecture

4.1 Overview

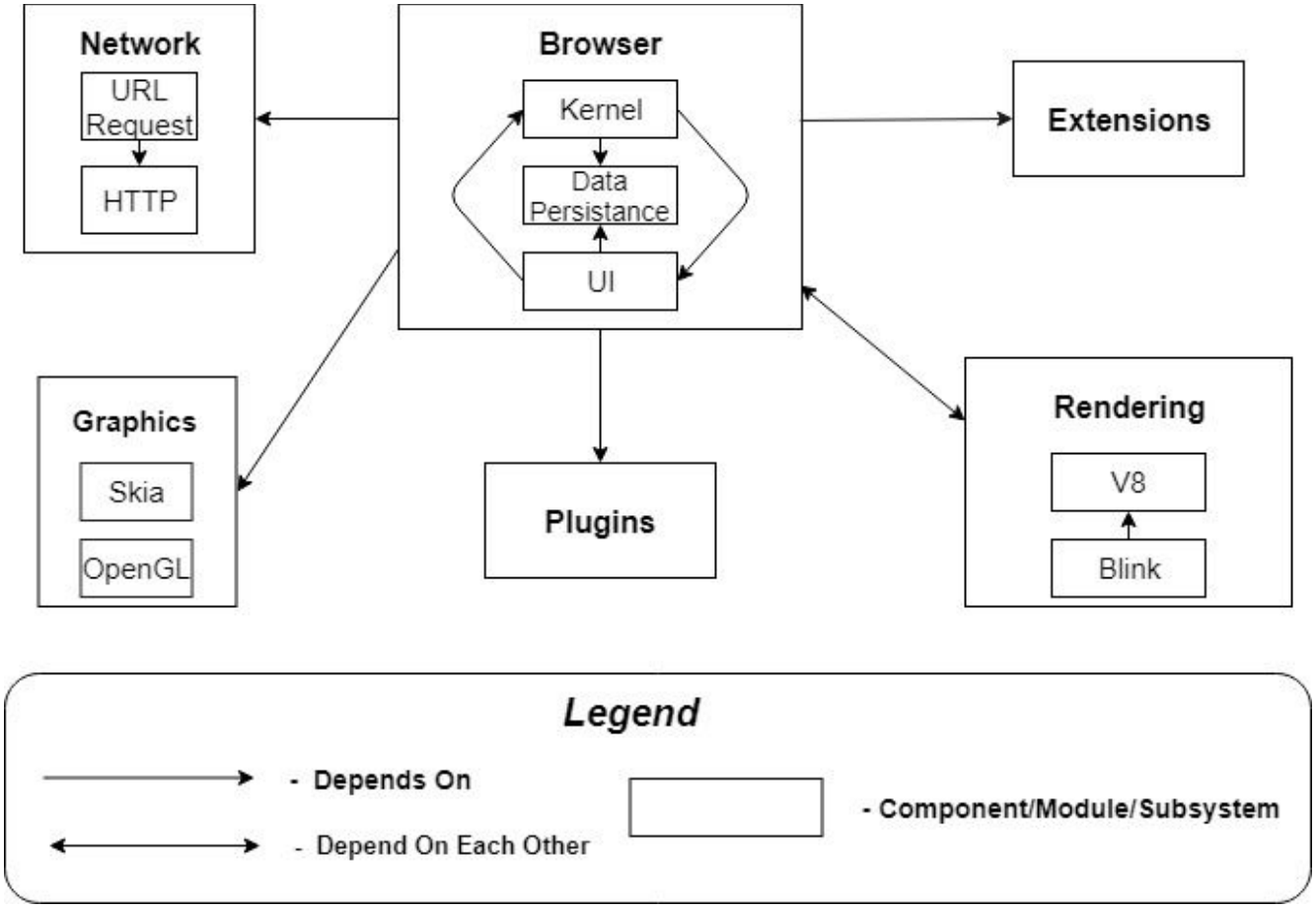


Figure 3. Finalized conceptual architecture diagram

Our final architecture is shown in Figure 3. This takes into account Chrome's multi-process architecture and sandbox model for security. Extensions became its own subsystem, and the kernel was added as a module to Browser from our finding in section 3.1. We also found that

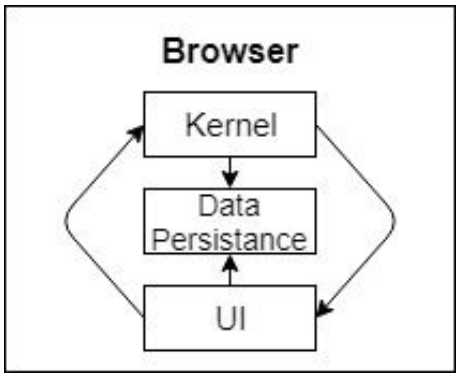
Blink is the primary rendering engine of Chrome, and added this to rendering. In addition to Blink, we felt V8 wasn't major enough to belong in its own subsystem, so we included it inside Rendering. Chrome has a range of subsystems will be discussed below.

4.2 Subsystem Breakdown

4.2.1 Browser

The browser subsystem is at the core of our whole architecture, linking between all other subsystems and processing any requests which have been made. This is achieved with the aid of the three main modules: kernel, data persistence and UI.

Data Persistence: All data such as sign in details, bookmarks and cookies are stored internally here. Data Persistence can also use the network subsystem to sync data through the cloud as we will see in a sequence diagram later. Data persistence is the only module which can function without having to rely on other modules within the browser subsystem .



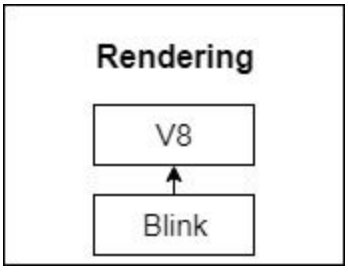
User Interface (UI): As the name suggests this is where all user input is handled. The UI depends on both of the other modules. It depends on data persistence so it is able to sign in as well as display content such as bookmarks on tabs. UI must also rely on the kernel so user input such as the pressing of a key or the movement of the mouse can be sent to the kernel for processing and distribution.

Kernel: This is considered the "brain" of the browser. All data marshalling between other subsystems is handled here, thus effectively sandboxing the whole environment. Kernel depends on data persistence as well as the UI. Kernel must depend on data persistence so it is able to load content such as extensions. The kernel must also depend on the UI so that it is able to return the result of a user input if it has an effect on the interface. An example of this could be if Javascript (running in a renderer) opens a new tab, this command is sent to the Kernel, which must then notify the UI.

4.2.2 Rendering

The Rendering subsystem's function is to parse and render web pages. It contains 2 modules which support this functionality. These are: Blink and the JavaScript V8 engine. Blink is a rendering engine, which is responsible for rendering HTML and CSS web pages. It, in turn, relies on the V8 engine to run any JavaScript that a given page uses.

Blink: Blink is a rendering and layout engine, responsible for rendering web pages to display to the user. It was created as a fork of the WebCore component of the Webkit engine, which is used in browsers like Safari. Blink is developed separately from Webkit and



contains support for the Document Object Model (DOM) which enables JavaScript manipulation of HTML, and Scalable Vector Graphics (SVG) [5]. Blink relies on the V8 engine for executing JavaScript code, which is an important part of most modern websites.

V8 Engine: The V8 engine is a high-performance JavaScript engine initially developed for Chrome. It improves performance over traditional JavaScript interpreters by compiling JavaScript code directly into native machine code before execution. This contrasts with traditional approaches such as interpreting bytecode. Additional optimization is done dynamically at runtime, through a variety of methods including inline expansion and inline caching [6].

4.2.3 Graphics

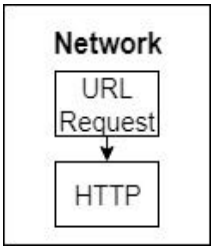
Graphics is a subsystem for drawing graphics to the user's device display. This is different from Rendering, which is at a higher level and is more abstracted. Graphics contains the Skia graphics library, which contains APIs for drawing to numerous target devices.

Skia: Skia is an open source 2D graphics library used in Chrome, the Android OS, and Firefox, among others. Skia enables Chrome to draw shapes, text, lines, and images. Skia's extensive API enables effective graphics drawing across many platforms and devices [7].

OpenGL: OpenGL, or Open Graphics Library, is the primary library used by Chrome to render 3D graphics.

4.2.4 Network

The network subsystem handles all networking functions of Chrome, such as communicating with servers and transferring files. It employs two modules for these tasks: URL Request, and HTTP.



URL Request: URL Request is responsible for handling requests to and from external servers. It keeps track of the various requirements that might be needed to fulfill a given request, such as cookies, host resolver, proxy resolver, or cached data [8]. It relies on the HTTP module to fulfill requests that require communication over HTTP.

HTTP: Hypertext Transfer Protocol is the primary application protocol of the internet, and is at the core of any web browser's network communications. The HTTP module is used by the URL Request module to handle all HTTP traffic.

4.2.5 Extensions

Extensions contain the application programming interface (API) for third-party developers to use for writing browser extensions, small programs that affect the user's browsing experience. Browser extensions such as ad blockers and password managers have become widely used today,

and extensions are a significant part of nearly every major browser. Since extensions typically require data to function, the Extensions subsystem also provides the ability for an extension program to interact with and modify received HTML, CSS and JavaScript code.

4.2.6 Plugins

Plugins are similar to “Extensions” but it is more integrated into the browser’s core functionality and is far more complex than browser add-ons. The main difference between the two is that plug-ins provide extra functionality which does not modify the core functionality. It is a third-party program that is plugged-in to a web page and affects only the web page that is using the plugin. Examples of plugins would be Flash or a PDF reader. Essentially, the biggest difference between Plugins and Extensions is that plugins are standalone programs that integrate into Chrome whereas extensions are programs that can only be used in Chrome.

4.3 Subsystem Dependencies

There are many dependencies within our conceptual architecture diagram, however almost all are dependencies required by the Browser subsystem.

Browser → Network dependency: No web browser would be useful without the ability to communicate with other machines through the internet, and Chrome is no exception. Browser relies on Network for all its networking operations, such as transferring files and communicating with servers via HTTP.

Browser → Extensions dependency: Browser relies on Extensions in order to perform operations on received HTML/CSS/JavaScript code. If the user has an extension installed that removes advertisements, for example, Browser would need to work with Extensions to make the required changes to the code it received before sending it to Rendering.

Browser → Plugins dependency: Browser will call plugins as necessary if a plugin has previously been determined to be required by Rendering earlier in the process. Essentially, when Rendering reports back to Browser with the results of parsing the HTML code it was given, Browser will be told whether or not a plugin like Flash will need to be loaded to properly render the page. Browser will then call plugins to handle this.

Browser → Rendering dependency: Browser relies on Rendering to parse and interpret HTML and CSS and run JavaScript code that it receives from the internet.

Rendering → Browser dependency: We decided on a dependency between Rendering and Browser so that JavaScript code changes, and resource requests could be handled by the Browser more quickly. The Browser likely wouldn’t ask Rendering every few milliseconds if there were JavaScript changes, since that could impede performance, so the V8 engine would need to be able to communicate directly with Browser to inform it of anything it needs to know. When a Renderer needs an additional resource from the network such as an image, the Renderer needs to request it from the browser.

Browser → Graphics dependency: Since the UI module inside Browser requires a method to draw the UI of Chrome on the user's screen, Browser relies on Graphics. Also, we decided not to put any dependencies between Graphics and Rendering to limit coupling and to improve the security of the system, as coupling has been shown to increase security vulnerabilities in software like Chrome [10]. Renderers would thus also access Graphics via the Browser module, sending and receiving graphics related commands.

4.4 Sequence Diagrams

Figure 4 illustrates the use case where a user requests a page that contains Javascript and having it rendered. For the user to view any given web page, a user must first type a URL. Browser sends this request to the URL Request module to handle. Then URL Request sends request to HTTPS module to fetch any required data from the website server, then sending it back to Browser. When requested data (HTML files, CSS files, etc.) is received, Browser sends all code to Extension. Extension returns a modified copy of the data back to the browser if any relevant Chrome extensions are installed. After Browser receives the modified data from the Extensions subsystem, it is sent to the Rendering subsystem. Within Rendering, the Blink engine processes HTML and CSS code while V8 executes the JavaScript part of the code.

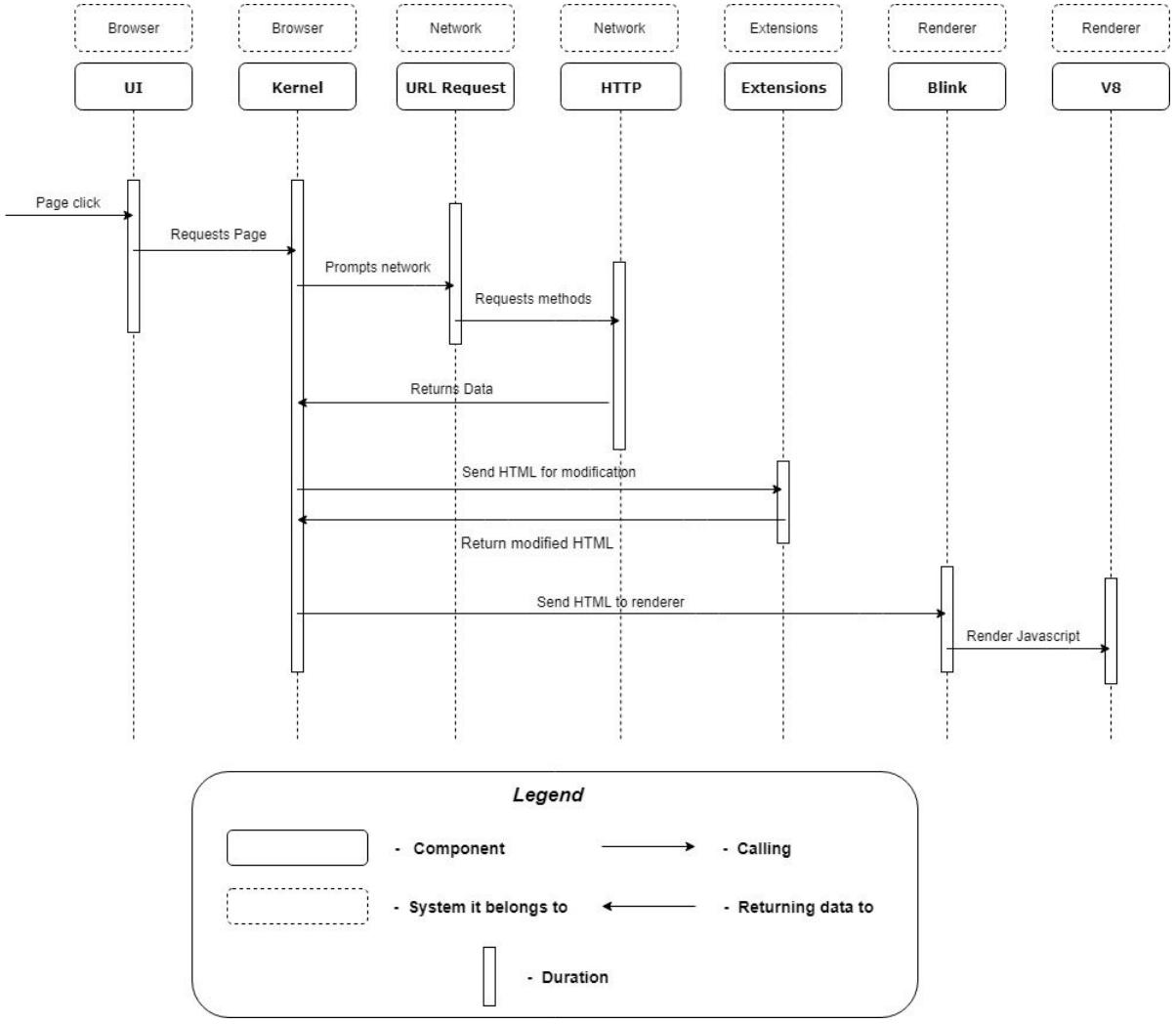


Figure 4. Sequence diagram showing the case of a user requesting a page that contains Javascript and it being rendered.

Figure 5 illustrates the use case where a user logs into a website and Chrome asks whether or not to save their login information. When a user signs into a website, UI recognizes a click event from the user clicking the login button and sends that to Kernel to handle. Kernel then sends this information to the Blink rendering engine to check what action needs to be taken to submit the login form to the website. Blink returns the necessary information, passing it back to the Kernel. Kernel recognizes a login was attempted and tells UI to display a dialogue box asking whether or not the user wants to save the login information for the website. If the user presses yes, then the login information is passed to Data Persistence to be stored locally. When the local save is finished then this information is passed to the network to be synced with Google's cloud system.

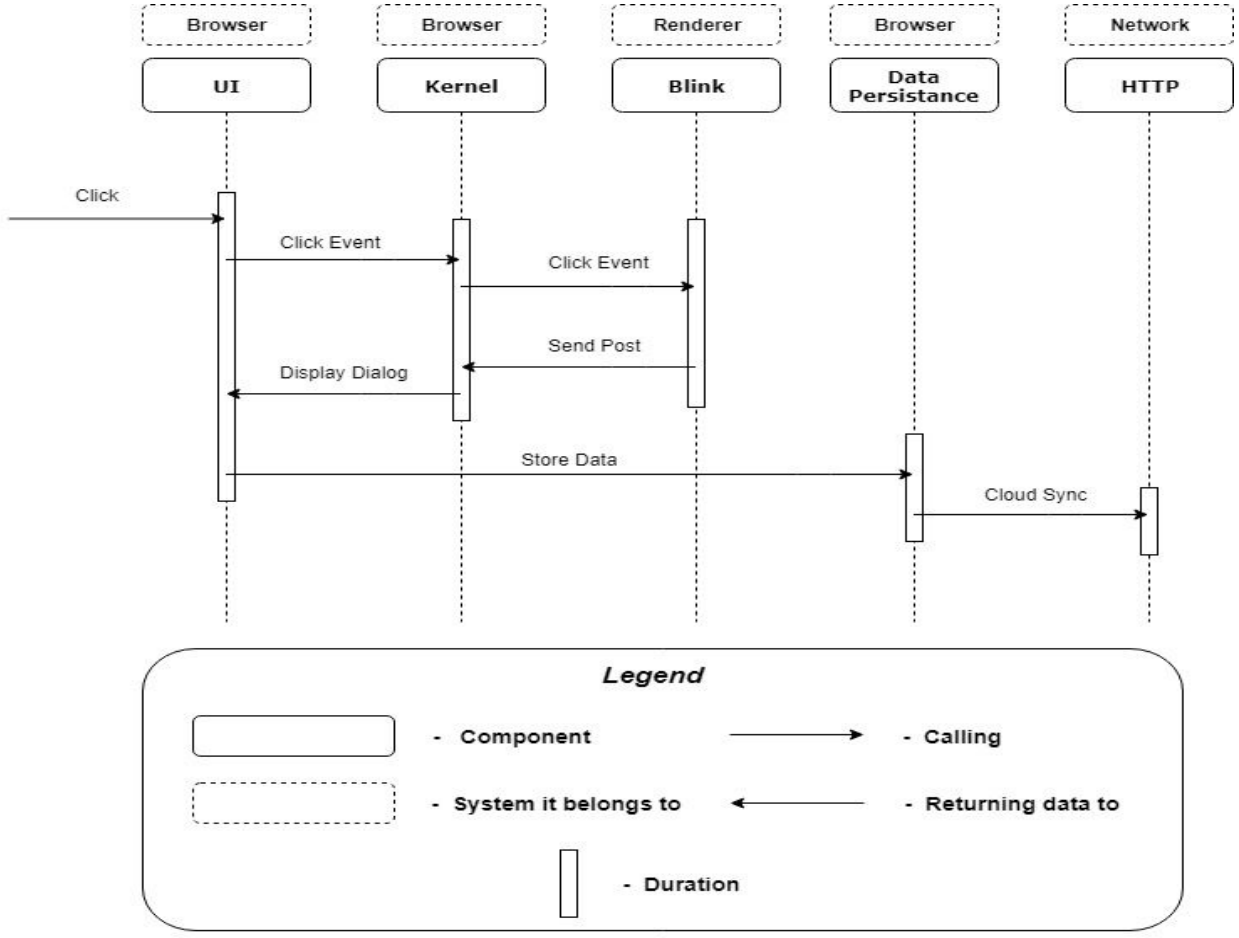


Figure 5. Sequence diagram showing the case of a user signing into a website and having their credentials stored by Chrome

5.0 Limitations and Lesson Learned

Throughout the duration of this assignment, our group encountered multiple limitations that brought on many challenges.

- One of the initial problems we encountered was a considerable scarcity of detailed architecture examples. As Google Chrome itself is not open source, we had to rely on unofficial sources and the Chromium project website (which was not always up-to-date) in order to get the necessary information. By relying on only a few sources, it was difficult to get a better understanding on certain subjects, such as determining dependencies and data flow. It was also a challenge to find material on the team structure of Chrome since Google doesn't release much information. Even chromium's website has little information on the team structure.
- Chrome has also evolved much in the last 10 years so much of the documentation or examples that we looked at were not up to date. Thus, we could not apply those directly

to our projects and it was difficult to distinguish what information is current and which is not.

That said, we also made several mistakes and learned valuable lessons while working on this project.

- We learned why architecture is such as specialized task done by experienced architects and engineers, as they need to figure out a wide range of functions and how to structure components in an efficient and secure way.
- We first assumed plugins and extensions are the same. For example, we thought extensions such as Adblock were considered to be plugins, however, that is not the case. While plugins like flash are slowly being phased out by Chrome [9] users are still using them. The Chrome team is prioritizing security more highly now than in the past and recognizes that plugins pose a significant security risk.
- We also found that it hard to figure out what the browser component does, as there were many conflicting definitions online. Our team also initially believed that the v8 engine was an external component since it was used in other software such as Node.js, however we came to realize that it is actually packaged with Chrome.
- Chrome is a massive topic with many articles and definitions online. Thus, we need to know when we collected sufficient information to start our analysis.

6.0 Conclusion

In summary, Google Chrome is designed around an Object-oriented architecture, supported by implicit invocation for inter-process communication. The browser acts as a central orchestrating subsystem that controls many Renderer instances and marshals data between subsystems. Chrome's multi-process architecture enforces strict security rules upon the browser ecosystem while promoting stability.

We had some issues early on in our derivation process, and found it difficult to find reliable sources, but in the end we believe we have settled on a fairly accurate and up to date model. We came to the conclusion of our architecture by looking at the core functionalities (locally stored data, HTML rendering, Javascript, extensions) and core non-functional requirements (concurrency, stability and security), and building around them. Ultimately our final architecture has low coupling with few connections between the various subsystems , and aims to maximise cohesion within each subsystem by grouping similar subsystems and functions together.

It will be interesting to see how well our conceptual architecture relates to the concrete architecture of Google Chrome. We expect there will be much less favourable coupling in the concrete architecture, especially between the Renderer and Graphics subsystems in the interest of speed. We look forward to researching all this in more as the project progresses.

7.0 References

1. Core Principles. (n.d.). Retrieved October 18, 2018, from https://www.chromium.org/developers/core-principles
2. Chromium Blog: Multi-process Architecture. (2008, September 11). Retrieved from https://blog.chromium.org/2008/09/multi-process-architecture.html
3. Deng, Z. (2018, March 21). Explore the Magic Behind Google Chrome. Retrieved October 18, 2018, from https://medium.com/@zicodeng/explore-the-magic-behind-google-chrome-c3563dbd2739
4. Threading and tasks in chrome (n.d.). Retrieved October 18, 2018, from https://chromium.googlesource.com/chromium/src/+/master/docs/threading_and_tasks.md
5. WebKit. (2018, September 17). Retrieved October 19, 2018, from https://en.wikipedia.org/wiki/WebKit
6. Chrome V8. (2018, September 24). Retrieved October 19, 2018, from https://en.wikipedia.org/wiki/Chrome_V8
7. Graphics and Skia. (n.d.). Retrieved October 18, 2018, from https://www.chromium.org/developers/design-documents/graphics-and-skia
8. Network Stack. (n.d.). Retrieved October 18, 2018, from https://www.chromium.org/developers/design-documents/network-stack
9. Li, A. (2017, July 25). Google sets roadmap for phasing out and removing Adobe Flash from Chrome by 2020. Retrieved from https://9to5google.com/2017/07/25/google-chrome-adoble-flash-end-plan-2020/
10. Lagerström, R., Baldwin, C., Maccormack, A., Sturtevant, D., & Doolan, L. (2017, March 21). Exploring the Relationship between Architecture Coupling and Software Vulnerabilities: A Google Chrome Case. Retrieved from http://nrs.harvard.edu/urn-3:HUL.InstRepos:30838136
11. Woo, H. (2016, May 31). Javascript Engine & Performance Comparison (V8, Chakra, Chakra Core). Retrieved from https://developers.redhat.com/blog/2016/05/31/javascript-engine-performance-comparison-v8-charkra-chakra-core-2/