# CISC 322/326 Fall 2018: A2 Report

Google Chrome Concrete Architecture Analysis

November 9th, 2018


TEAM 25: CHROME CASTAWAYS

Erik Koning

Roman Sokolovski

Ryan Rossiter

Sammy Chan

Sarah Nowlan

Scott Reed

# Table of Contents

# Abstract

In this report we will be analyzing the concrete architecture of Google Chrome web browser, we do so by looking at the source code and using the Scitools Understand program. Firstly, we investigated the Chromium directories and decided how they would best fit into out conceptual architecture and subsystems. We made minor change to our conceptual architecture but kept our object-oriented style and most of the subsystems with their dependency. Once we got our final conceptual architecture and our concrete architecture, we examined all the dependencies that were different between the two architecture and explained why they were altered. We then investigated the subsystems that we have in our concrete architecture and compare them to their architecture in our conceptual and what changes have been made to the subsystem. The report than uses sequence diagrams to show how the dependencies work and who processes play our in our architecture. Next, we expressed the limitations we had while working with the source code and possible issues the Google Chrome development team would encounter whil trying to implement our concrete architecture. Lastly, this report covers the lessons we learned doing this assignment and gives a brief introduction to the proposed feature we will be investigating in the next assignment.

# Introduction

Concrete architectures are important so the specific components necessary for implementation can be revealed and shown in a manner that can be followed like a map to determine how the system will accomplish tasks. There are many various relationships in large software systems that need to be revealed so the components usability, weaknesses, and strengths can be realized [3]. There exist programs to aid in the discovery of hidden dependencies between system such as scitools-understand software. Such software operates by examining each file in a system (such as the networking sub-system) and inspects each method to see if a file from a different sub-system is being called upon, if there exists such a method call it is considered a dependency.

In our previous report titled: Google Chrome Concrete Architecture Analysis, we started by looking at Chrome and Chromium as a whole and identified the tasks it preforms to operate as a browser, we then categorized each task into subsystems, and iterated from there. While looking at the dependencies of these subsystems we settled on an object-oriented style to be most fitting for our conceptual architecture. Through iteration and challenging of our original conceptual architecture, we have derived our finalized conceptual architecture, and will be further testing its validity by investigating the source code of chromium. This report will discuss our finalized conceptual architecture, the concrete architecture we have analyzed, and an in-depth look at how two different subsystems function.

# Original Conceptual Architecture

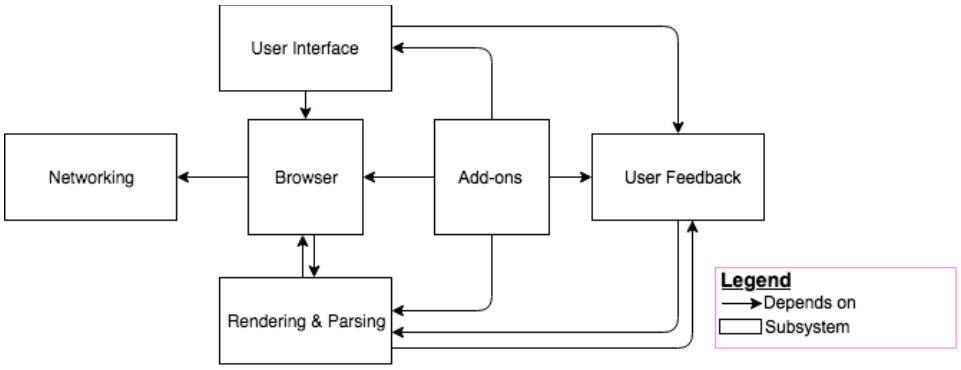


Figure 1: Initial Conceptual Architecture

Our initial conceptual architecture was an object orientated architectures consisting of six subsystems. We believed initial object orientated architecture fit well with chromium's design and did not feel any revisions were needed. During our derivation process to decide which chromium directories belonged within which subsystems of our architecture, we found none fit within the Add-on subsystem and removed it. Aside from this change, no further alternatives were investigated.

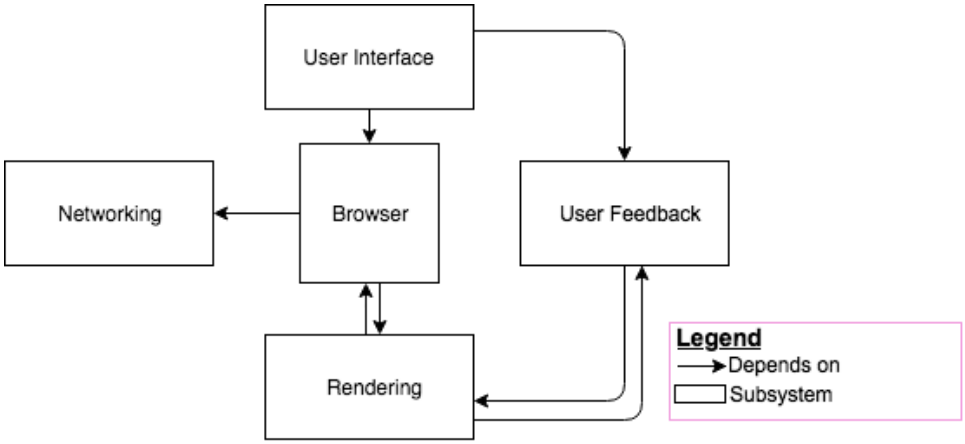


Figure 2: Updated Conceptual Model

# Derivation Process

Using Scitool's Understand tool, we observed and analyzed the directories comprising the chromium architecture. From investigating the various directories, we attempted to identify the basic functionality each directory was capable of and decided which subsystem the directory would best fit within our conceptual architecture. In doing so, we discovered many of the functions we believed would be separate, chromium has grouped together within the same directories and folders. As a result, while mapping directories we found many of them to be

grouped up into our Browser subsystem, while most of the other subsystems having only one directory within. The Add-ons subsystem was found to have no directories that would match well within it, and upon further research discovered chromium handles items such as plugins, extensions and apps within the Components directory along with other functionality, rather than dedicating a separate subsystem to them. For this reason, the Add-ons directory was removed from our conceptual architecture. Further separation and division and decomposition of items within the directories could have resulted a model more fitting for our conceptual but due large number of sub-directories within each, we decided to leave them as they were. Scitool's Understand then calculated the dependencies, the differences which can be observed within our reflexion model.

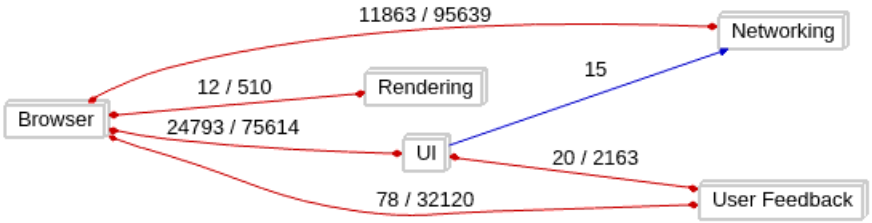


Figure 3: Dependency Graph generated by Understand

## Concrete Architecture

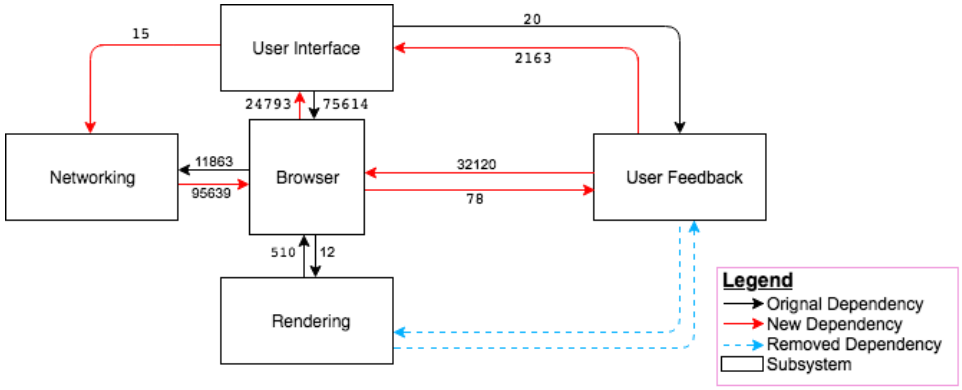


Figure 4: Reflexion Model with dependencies calculated by Understand

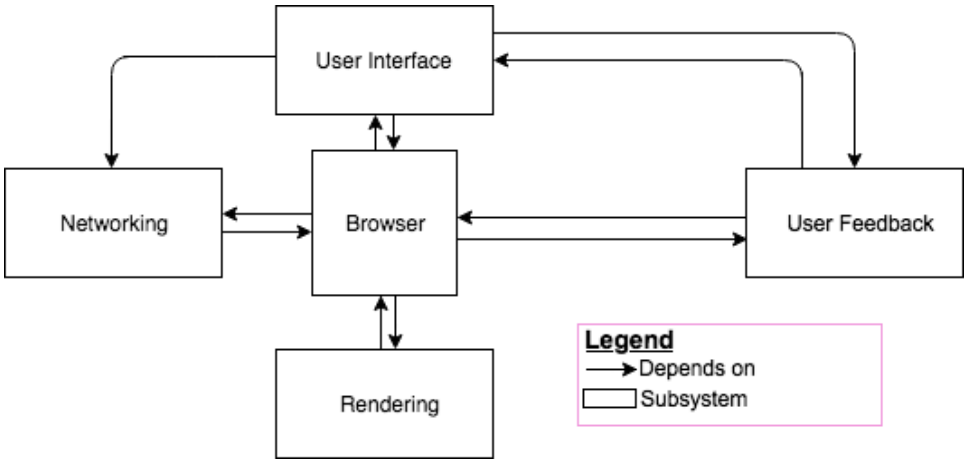


Figure 5: Concrete Architecture Model

## Components/Subsystems

### Browser

The Browser subsystem contains most of the project directories, since many of them are tightly coupled together. It includes /chrome, /content, /components, /services, /storage, /sql, /base, and /mojo. As a result of having all these directories, Browser possesses bidirectional dependencies between all other subsystems. Features that would have been in the Add-ons subsystem in our conceptual architecture, such as plugin, extensions and apps are managed by this system. The Browser subsystem is essentially the core the system, responsible for managing data and basic functionality.

### Networking

This subsystem contains all code and utilities required for the browser to communicate with the network. It consists solely of the /net directory. Networking has a bidirectional decency between it and Browser. Browser depends on Networking to retrieve data from the network while Networking requires information about the browser.

### Rendering

Conceptually, the rendering subsystem is responsible for rendering a page's content. This operation is performed using Blink for the most part, but since the Blink interface is contained in Browser/content we found it difficult to separate it into Rendering. In our concrete architecture, the Rendering subsystem only contains the /gin directory. Had we performed a more thorough decomposition of the subdirectories, we would expect our concrete architecture's Rendering subsystem to more closely match our conceptual.

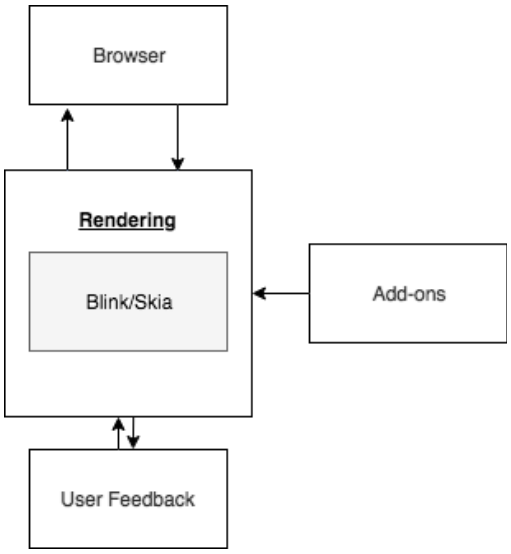


Figure 7: Rendering Conceptual Architecture

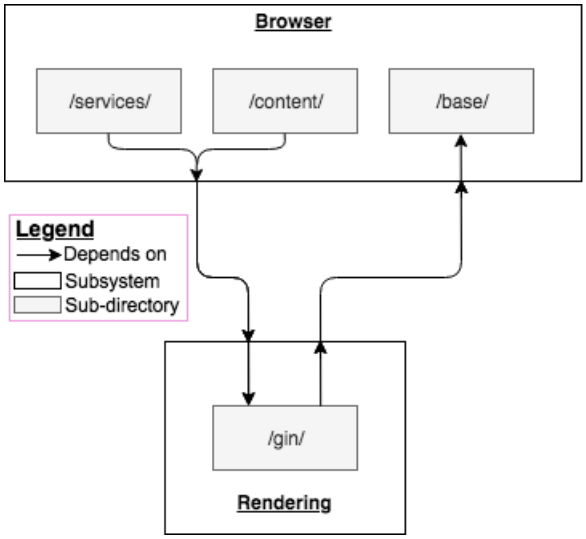


Figure 6: Rendering Concrete Architecture

*Note only dependencies, subsystems and directories relevant to Rendering are shown

Just as with our conceptual architecture, there exists a bidirectional dependency between Rendering and Browser. With Rendering containing very little, and much the rendering features being moved to Browser, the dependencies between Rendering and User Feedback have been removed along with the Add-ons subsystem as it was removed in its entirety. The /gin directory contains primarily files relating to v8, Chromium's open source Javascript engine. This would be used for compiling and rendering Javascript on web pages.

## User Interface

As one would expect, the User Interface subsystem contains the /ui/ directory. This subsystem is responsible for aspects of the browser such as, the browser interface and window appearance and buttons on the interface. It also contains gfx and gl which are shared graphics libraries which are utilized by some of the other subsystems as well. The User Interface primarily depends on the browser and Browser and User Feedback subsystem just as it did in our conceptual model, with some minor dependencies on Network.

## User Feedback

In our conceptual model, we viewed the User Feedback subsystem to be responsible for all types of user feedback, including display of webpages, browser interface, sound, input event reactions, etc. In the concrete architecture, many of these features were grouped within other directories with other non-feedback files, rather than isolated by themselves. As a result, in our concrete architecture, User Feedback only contains the /cc directory, which is the Chromium Compositor. It handles user inputs such as scrolling & selecting, animations, colour, and divides a web page into layers which allows it to be viewed more smoothly despite rendering stalls.

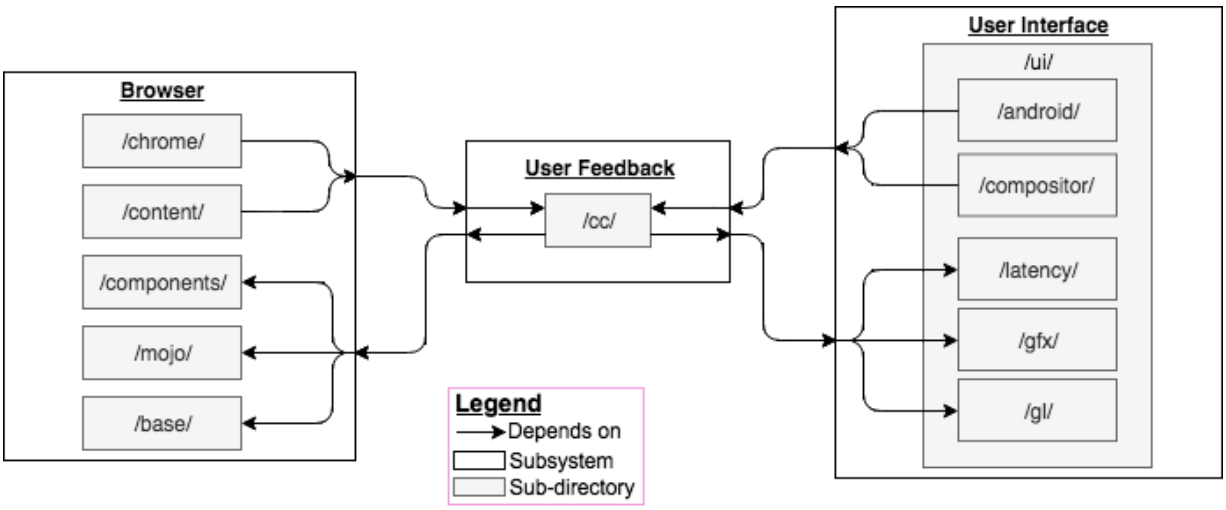


Figure 8: User Feedback Subsystem Concrete Architecture

*Note only dependencies, subsystems and directories relevant to User Feedback are shown

## Reflexion Analysis

Our reflexion analysis process involved identifying the discrepancies between the dependency graph generated by Scitool's Understand and our updated conceptual model. Comparing the two, our relfexion model was made to highlight the difference between the two. Within our reflexion model, we observed four major discrepancies and a single minor discrepancy, between our conceptual and concrete architecture models. The first major discrepancy is the removal the bidirectional dependencies between User Feedback & Rendering being replaced by new bidirectional dependencies between User Feedback & Browser. The other major discrepancies additional unidirectional dependencies, resulting in bidirectional dependencies in what we considered unidirectional dependencies.

The reason for the new dependencies between User Feedback and Browser rather than Userfeedback and Rendering, is that we found that chromium primarily uses third-party resources for rendering, which were not included in the directories used for Scitool's calculations. Additionally, many of the rendering tools are contained within directories along with common and basic browser components. Since much of the rendering tools were placed within the Browser subsystem instead, it is only natural for the dependency to have shifted as well. Some of the most notable dependencies from the chrome/browser folder and files pertaining to android and vr being dependent on the /cc Chromium compositor within rendering. The Chromium compositor helps load a page in layers for smoother viewing without rendering stalls. The nature of android and vr devices would require their own specialized method of doing so, so they require files specific to them in both directories. When developing our conceptual architecture, we neglected to consider its function on other platforms and what they would need to run. The content directory also has its own renderer which relies on the compositor as well.

From the reflexion model, a new dependency from Network to Browser is observed. Previously in our conceptual architecture, there was only a single unidirectional dependency from Browser to Network but that has now become a bidirectional dependency. The reason for this dependency appears to be caused items known as a network list observer and QUIC. There is little documentation on the network list observer but what little information present seems to indicate it is sued in peer to peer connections, with the network possessing a notifier to alert the browser in any change of connection. QUIC is a transport protocol developed by Google in order to reduce latency, dependent with communicating items in both the /components and /net directories. Though in our conceptual architecture the dependency between Browser and Network was viewed as unidirectional, it appears that these systems are more interdependent than we anticipated and bidirectional relations was needed for optimizing performance.

A new dependency from Browser to User Interface is formed as a result of the various directories within Browser being dependent on the shared graphics library /ui/gfx, along with many other items. Again, we initially viewed this relation as unidirectional, but the systems were more interdependent than we could have known. One example of a dependency from Browser and gfx would that data concerning bookmarks stored within the Browser requires use of the gfx library to determine favicon size. Additionally, since apps are managed by the browser rather than its own separate system as we initially conceptualized, the /chrome/app folder requires access to the User Interface in order to modify it if needed.

User Feedback has also gained a new dependency on the User Interface we did not originally foresee. The cause of this discrepancy can be attributed primarily to the Chromium compositor's dependency on the gfx library found within the User Interface. Items used within the gfx library relate to things such as the geometry of the browser, animations, input effect and layering should be done. This is understandable given many webpages' display may have some aspects that reflect the browser and operating system one uses. With our conceptual architecture, we believed a single system to be responsible for rendering display of both the graphical user interface and the web pages themselves but in reality, the User Interface manages its own display.

The final discrepancy is a minor one which could possibly be ignored entirely. It is a dependency between the User Interface to the Network subsystem, the number of dependencies themselves consisting of only fifteen. This appears to be caused primarily by the /ui/base/ directory along with two dependencies from /ui/aura. The main subdirectory utilized for its feature is the net/base/ subdirectory for simple features such as filename_util.h and escape.h which convert a URL to a filename and escape characters respectively.

# Use Case Sequence Diagrams

## Logging into a Website and Chromium Saves the Password

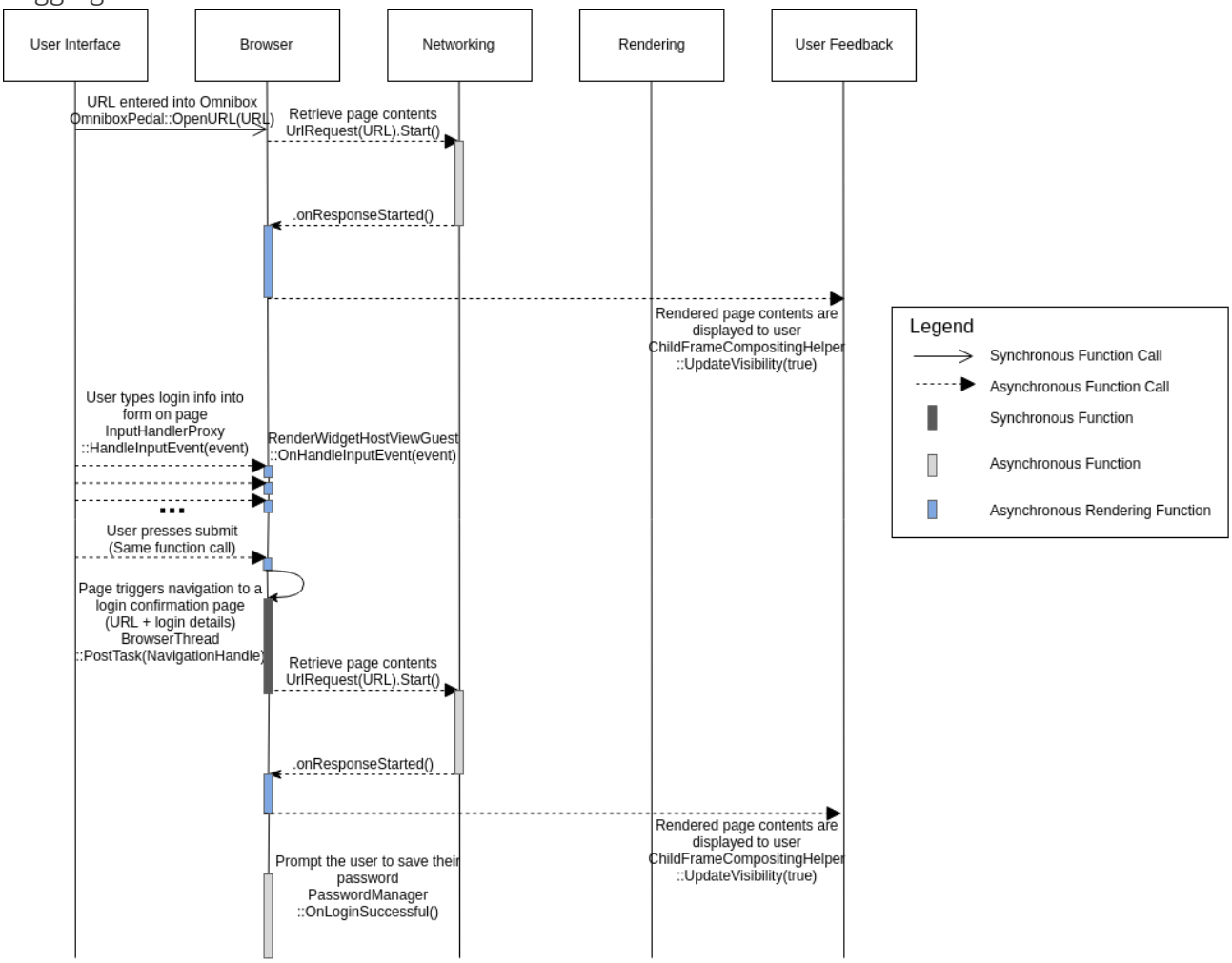


Figure 9: Logging into a website with credentials - Sequence Diagram

In this sequence diagram, a user enters URL into the Omnibox, the OmniboxPedal::OpenURL(URL) function is invoked, notifying the browser to begin loading a new page. The browser begins loading the page by initiating a UrlRequest(URL).Start(). Once the Browser is notified that the content was loaded from a .onResponseStarted() event, it passes the content to Blink (within Browser/content) to be rendered. Once there is rendered content t to be displayed, the RenderFrame is shown to the user with a call to ChildFrameCompositingHelper::UpdateVisibility(true). Once the page has loaded, the user types their login information into the form on the page, which triggers a series of input events from the UI's InputHandlerProxy::HandleInputEvent(event) function. When the user presses submit, a page navigation is triggers with an IPC message in BrowserThread::PostTask(NavigationHandle). This in turn begins another UrlRequest and page render, and upon successful login (as indication by the HTTP status of the returned page after navigation) the user is promped to save their password by the PasswordManager when PasswordManager::OnLoginSuccessful() is called.

## Chromium Rendering a Page with JavaScript

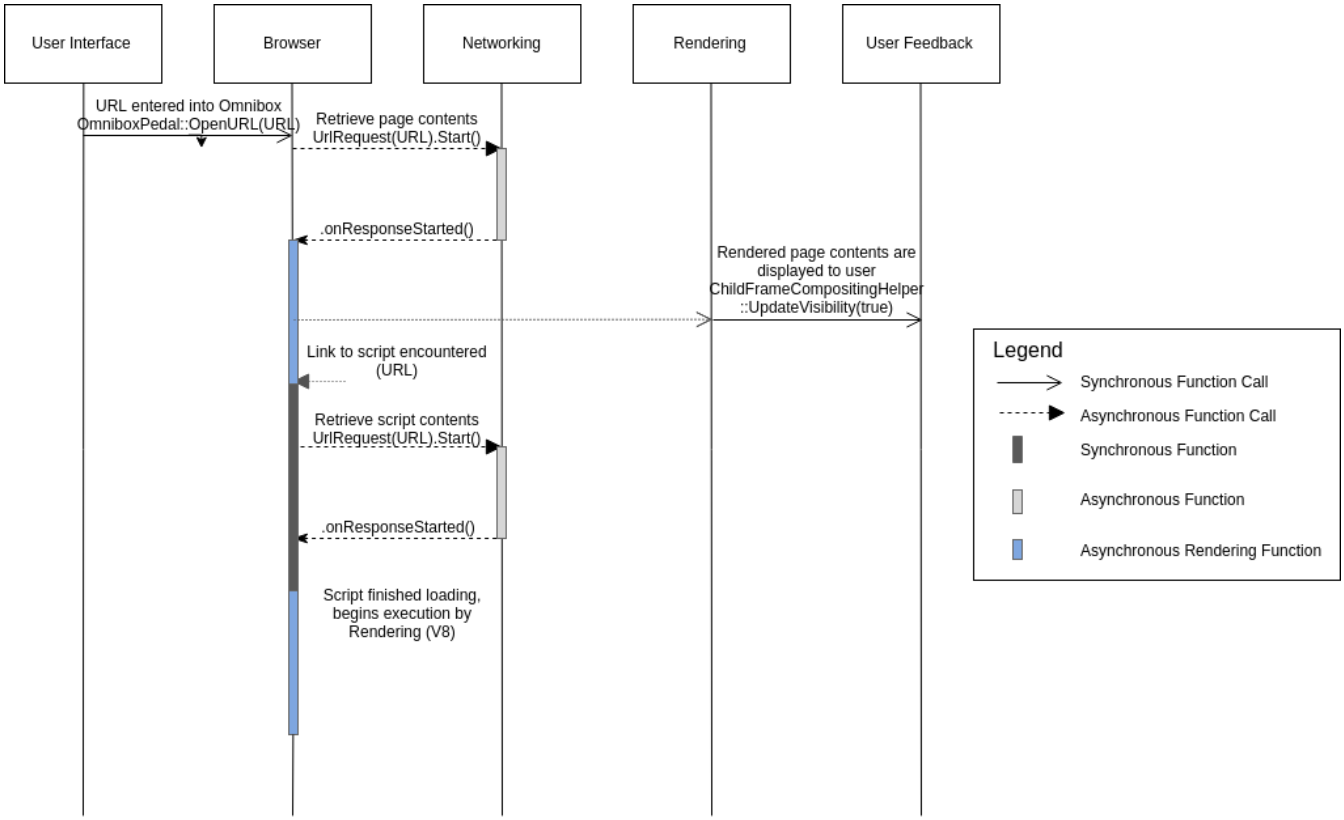


Figure 10: Rendering a JavaScript Page - Sequence Diagram

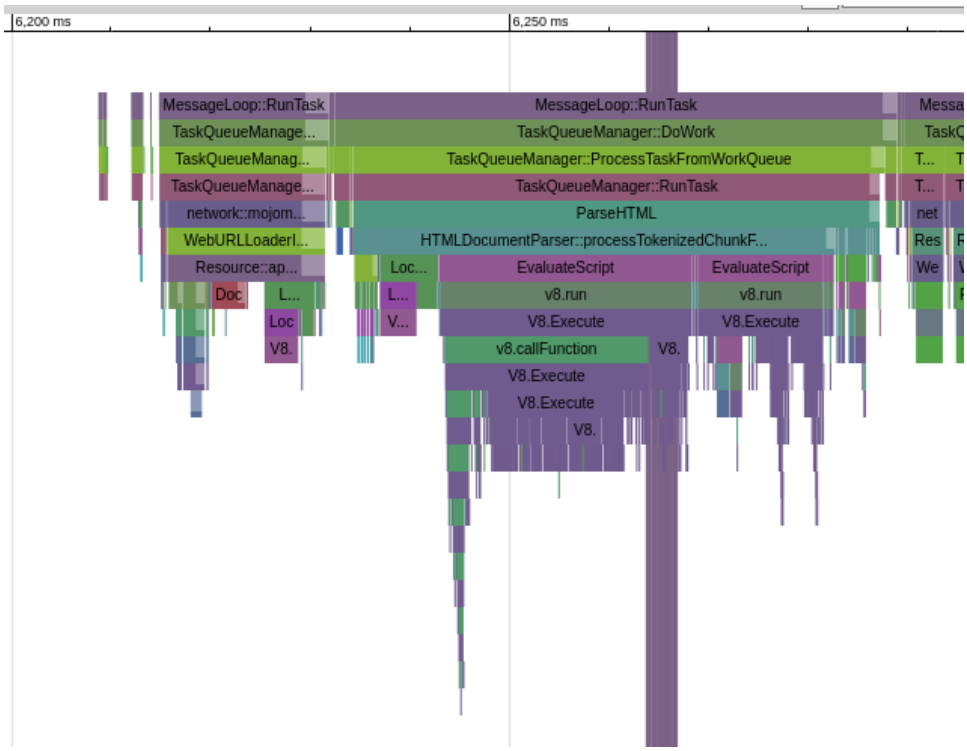


Figure 11: Captured Stack Trace of Rendering a Page using Chrome's Strack Tracing Tool

This sequence diagram illustrates the flow of execution within the browser when it is loading a page that contains a script to be run. When the user enters URL into the Omnibox, the OmniboxPedal::OpenURL(URL) function is invoked, notifying the browser to begin loading a new page. The browser begins loading the page by initiating a UrlRequest(URL).Start(). Once the Browser is notified that the content was loaded from a .onResponseStarted() event, it passes the content to Blink (within Browser/content) to be rendered. Once there is rendered content t to be displayed, the RenderFrame is shown to the user with a call to ChildFrameCompositingHelper::UpdateVisibility(true). In this example, a script tag with a URL source is encountered while parsing the document, so the Browser initiates another UrlRequest from Networking, and ultimately returns the script contents to the V8 engine to be executed.

## Limitations of Findings

Chromium is a large system with many different parts of it we still do not understand or are aware of. Investigating every aspect of the system would be unfeasible given the resources we have. What we were able to research was but a small portion of Chromium as a whole, and often even those findings were severely limited. The source code of the system proved difficult to understand and frequently lacked comments or documentation. In the event that there was documentation about a files function, what that function does and how it did it was still left unclear and required further research. At times we had nothing go off of but the names of the functions and files themselves. It is very possible that we may have completely misinterpreted the purpose of some files. Even then, the specific dependencies we did investigate were but few

of many other dependencies. Being open source, many different styles of code were encountered, increasing the difficulty attempting to understand it. Attempting to identify who was responsible for a particular file did not produce many results, as the very few comments that were present would simply attribute it to the Chromium Authors.

## Team Issues

When implementing our design, the chromium development team may find a few things difficult or challenging. First, there are a lot of complex subsystems in our architecture such as networking or user interface modules. Due to this the development team may find that completing and maintaining all of these complex modules can be a challenge and would require a lot more engineers than they might have previously expected. These developers would have to work in collaboration with each other to design, develop and maintain the system, so team size can be an issue. Next, the chromium development team may find that maintaining our design is quite the challenge since our subsystems are quite co-dependent on each other to run smoothly. The development team would have to make sure that every small change they make to any subsystem wont consequently affect other systems in the long run. Furthermore, since chromium is an open source project, anyone can contribute to the codebase. Therefore the team is going to find that version control is crucial for the ongoing security of the system and would have to allocate engineers for dev-ops.

## Lessons Learned

Throughout this assignment we encountered many obstacles and tough decisions that in the end gave us a better understanding of the issue at hand. We found that an overall good understanding of all or most of the other architecture styles is crucial. The knowledge gained from looking into different architectures helped us decide on which one effectively portrays our goal at hand, which is why we ended up with the object oriented style. Also, we found that communication between team members is crucial for an assignment of this size. Co-ordinating with team members on when to meet up and how to split up work became essential in completing this project and because of the amount of research needed, everyone had to be working on something in order for everything to be completed. Moving along we also found that architectural details are very important and that a high level view of the system is not very beneficial to us. Chromium is quite a complex project that has many small details and many more interconnected subsystems than initially expected, looking at just the main modules or function calls isn't quite enough to understand what’s really going on behind the scenes. Attempts to conceptualize a system with low coupling and high cohesion proved too idealistic. This is where the understand tool came in handy, it helped us divulge the information under the base layer and gave us a much more thorough understanding of the system that we couldn’t quite get by just looking at subsystems. Lastly, we found that when we were developing our conceptual in assignment 1, many systems/features were not taken into consideration when developing the architecture. This in turn led to a few discrepancies in our assignment 2.

## Feature Proposal

We are proposing an Intranet 'teamviewer' solution for chrome browsers. The feature would enable a user to cast their tab, or browser to other users either with a link or inviting them via their Google account. Other users can then view the casted tab or browser as a tab window on their machine. Chrome already has the Google Cast SDK built in, so enabling such a feature would be straightforward and have minimal road blocks. Most other solutions stream the whole monitor which would not be compatible with the Google Cast SDK thus will be too resource intensive. Chromecast uses the mDNS (multicast Domain Name System) protocol to search for available devices on a Wi-Fi network. Therefore, compatible browsers could discover other browsers on the same network for display linking.

The networking and UI subsystem will be affected by this feature as mDNS is a network protocol and the Google cast SDK is enclosed in Chrome's UI sub-system. Our implementation plan for this feature is to focus on how Chrome already casts its screen to a Chromecast video and make the browser an applicable cast target. This feature will solve the need for a lightweight solution for information sharing within businesses.

## Conclusion

Chrome's source code was quite vast and took a while for us to fully grasp the total whole of it and figure out how to break up the directories to sort them into subsystems. After developing the concrete architecture from the source code and by using Scitool's Understand, we are more confident that the architecture style is object-oriented which we originally thought with our conceptual architecture. We did have to make some adjustments to our conceptual architecture by removing the add-ons subsystems when the source code didn't line up with our originally understand of the system. Once we sorted the directories and Understand displayed the concrete architecture, we learned that the subsystems were not as independent as once thought and created new dependency which we didn't not expect but once investigated were logical. By looking into the limitations and issues the development team could have with our concrete architecture, we gained a better idea of the struggles of creating architectures and the methods used to make a well-structured system.

By examining and understanding the concrete architecture of Google Chrome and by using Scitool's Understand we are able to understand the structure of Chrome more thoroughly. This deeper understanding will great help us when it comes to figuring out the implementation of our proposed feature in the next assignment.

## Appendix

| **Network** | **Browser** |
|---|---|
| Net/base | components/viz |

| | |
|---|---|
| Net/third_party/quic/tools | content/renderer/network_list_observer.h |

Table 1: Networking files and its dependencies in Browser

| **UserFeedback/cc** | **Browser/components** |
|---|---|
| cc/animation | components/sync/base |
| cc/base | components/ukm |
| cc/input | components/viz/client |
| cc/layers | components/viz/common/display |
| cc/mojo_embedder | components/viz/common/frame_sinks |
| cc/paint | components/viz/common/gpu |
| cc/raster | components/viz/common/quads |
| cc/resources | components/viz/common/resources |
| cc/scheduler | components/viz/common/surfaces |
| cc/test | components/viz/service/display |
| cc/tiles | components/viz/service/display_embed |
| cc/trees | components/viz/service/gl |

Table 2: Userfeeback files and its dependencies in Browser

| | |
|---|---|
| Browser/components | cc/animation |
| Browser/content | cc/input |
| Browser/services | cc/layers |
| Browser/storage | cc/raster |
| | cc/resources |
| | cc/tiles |
| | cc/trees |

Table 3: Browser files and its dependencies in Userfeedback

| | |
|---|---|
| Browser/chrome | ui/accessibility |
| Browser/components | ui/andoid |
| Browser/content | ui/aura |
| Browser/services | ui/base |
| | ui/compositor |
| | ui/display |
| | ui/events |
| | ui/gfx |
| | ui/gl |
| | ui/keyboard |
| | ui/latency |
| | ui/message_center |
| | ui/native_theme |

Table 4: Browser files and its dependencies in UI

| | |
|---|---|
| Cc/animation | ui/gfx/animation |
| cc/base | ui/gfx/codec |
| cc/benchmarks | ui/gfx/geometry |
| cc/input | ui/gfx/ipc |
| cc/layers | ui/gfx/test |
| cc/paint | Buffer_format_util.h |
| cc/raster | Color_space.cc |
| cc/resources | Gpu_memory_buffer.h |
| cc/scheduler | Native_widget_types.h |

Table 5: User Feedback files and its dependencies in UI

# References

[1]haraken (14 August, 2018). *How Blink Works*. Retrieved 09 October, 2018 from https://docs.google.com/document/d/1aitSOucL0VHZa9Z2vbRJSyAIsAz24kX8LFByQ5xQnUg/edit#heading=h.v5plba74lfde

[2]The Chromium Project(2018). *The Trace Event Profiling Tool*. Retrieved 08 November, 2018 from https://www.chromium.org/developers/how-tos/trace-event-profiling-tool

[3] Grosskurth, A. & Godfrey, M. W. (2005). *A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser*. Retrieved 09 October, 2018 from http://cs.queensu.ca/~ahmed/home/teaching/CISC322/F18/files/emse-browserRefArch.pdf.