Chrome: Concrete Architecture

Assignment 2 – Report

Nov 9th, 2018

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

Abstract

The team was to investigate the Chromium browser source code and propose a concrete architecture based on studied findings and the previously designed conceptual architecture. The object-oriented design, with its five main subsystems—networking, rendering engine, user interface, data persistence and browser engine—was retained for the concrete architecture. The networking subsystem connects to the internet using FTP and HTTP. The rendering subsystem, made of multiple components, parses HTML and CSS and prepares the Document Object Model (DOM). The user interface presents the page and allows the user to interact with the browser. The data persistence subsystem collects and stores continuous data and information from users. The browser engine represents the top-level browser window and acts as the system's main control center. Each subsystem is a separate object in the object-oriented design structure, with various dependencies on each other.

The Understand software tool was used to determine new dependencies that were omitted in the original design. There were seven main dependencies that were missing from the conceptual architecture: UI to networking, rendering engine to data persistence, browser engine to UI, rendering engine to networking, networking to browser engine, rendering engine to UI, and UI to rendering engine. Each of the dependencies were initially overlooked because it was assumed that communication between the subsystems would happen indirectly through the browser engine.

The object-oriented design makes for easy scalability and evolution. New subsystems can be added and tested individually before being implemented into the preexisting system. The division of work between teams is simplified by each subsystem being its own object. However, the increased dependency complexity of the concrete architecture can cause issues for the development team, as it is essential for developers to understand how changes to one subsystem can affect all dependent components.

The accuracy of the concrete design was limited by inaccuracies in the official Chromium documentation, inefficiency of the Understand tool and general time constraints for the assignment. Chromium's vast, complex source code made it impossible to investigate each system component in detail, and research findings relied on many assumptions.

Introduction and Overview

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

The report discusses the process of developing the concrete architecture from the preexisting conceptual architecture design in Assignment 1. The Architecture section provides details and explanations for the determined concrete architecture. Each major component is discussed, along with their interactions and data flow. The report covers various technical benefits—testability, scalability and evolution—of the determined architecture, and their implications for concurrency and software development. A reflexion analysis is done to determine and investigate gaps between the conceptual and concrete architectures.

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

The Conclusions section discusses the increased architectural complexity caused by additional dependencies in the concrete architecture, and how it can result in issues for the development team. More research will need to be done for the feature proposed in Assignment 3. The Lessons Learned section focuses on the team's experience dealing with unsatisfactory resources, software issues and time constraints, each of which contributed to a problematic design process for the concrete architecture. The team also found it difficult to divide work evenly throughout the research and design phases of the assignment. As it was important for each team member to have a general understanding of the entire system and its interactions, tasks could not be easily divided by the separate system components.

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

Architecture

Derivation Process

Figure 1: Conceptual Architecture System Dependency Diagram shows the conceptual architecture proposed in Assignment 1.

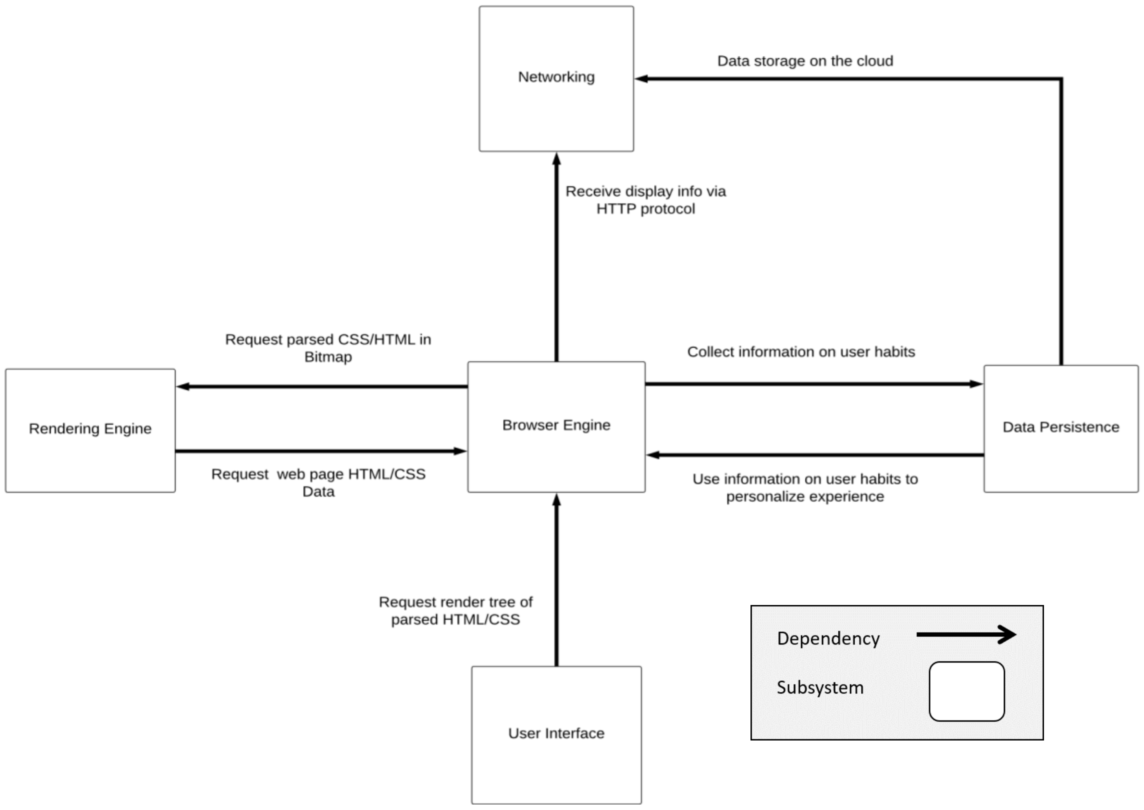

Networking
Data storage on the cloud
Receive display info via HTTP protocol
Request parsed CSS/HTML in Bitmap
Collect information on user habits
Rendering Engine
Browser Engine
Data Persistence
Request web page HTML/CSS Data
Use information on user habits to personalize experience
Request render tree of parsed HTML/CSS
User Interface
Dependency
Subsystem

Figure 1: Conceptual Architecture System Dependency Diagram

The team began Assignment 2 by investigating the provided Chromium source code and determined that each of the code folders could be successfully partitioned to fit under each of the existing five subsystems from their conceptual architecture. After some consideration, the team decided to stick with an object-oriented design style—with certain components having a layered structure internally—which allowed for the conceptual architecture to remain the same. In order to devise a concrete architecture, the team used the Understand software to build the architecture based on the source code.

The following is the division of source code folders to each of the subsystems: *ui* under the UI, *net* under networking, *cc, gin* and *content* under the rendering engine, *base, mojo, components, chrome* and *services* under the browser engine, and *sql* and *storage* under data persistence. The tool exposed multiple subsystem dependencies that had originally been overlooked. Figure 2: Understand Architecture shows each of the dependencies generated by Understand. The concrete architecture was created by adding the new dependencies to the conceptual architecture.

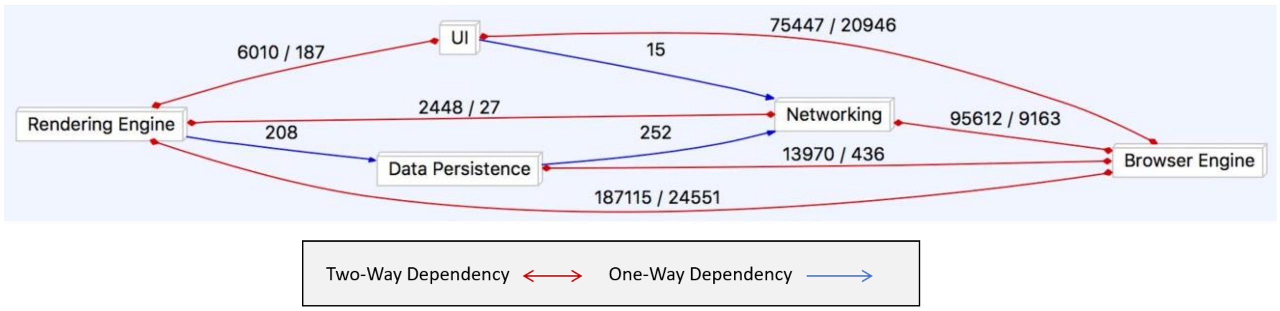


Figure 2: Understand Architecture

Major Components and Interactions

Concrete Architecture

The five main components in the concrete architecture, each organized as a separate object in the system, are the same as those from the conceptual architecture. They can be seen in Figure 3: Concrete Architecture.

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

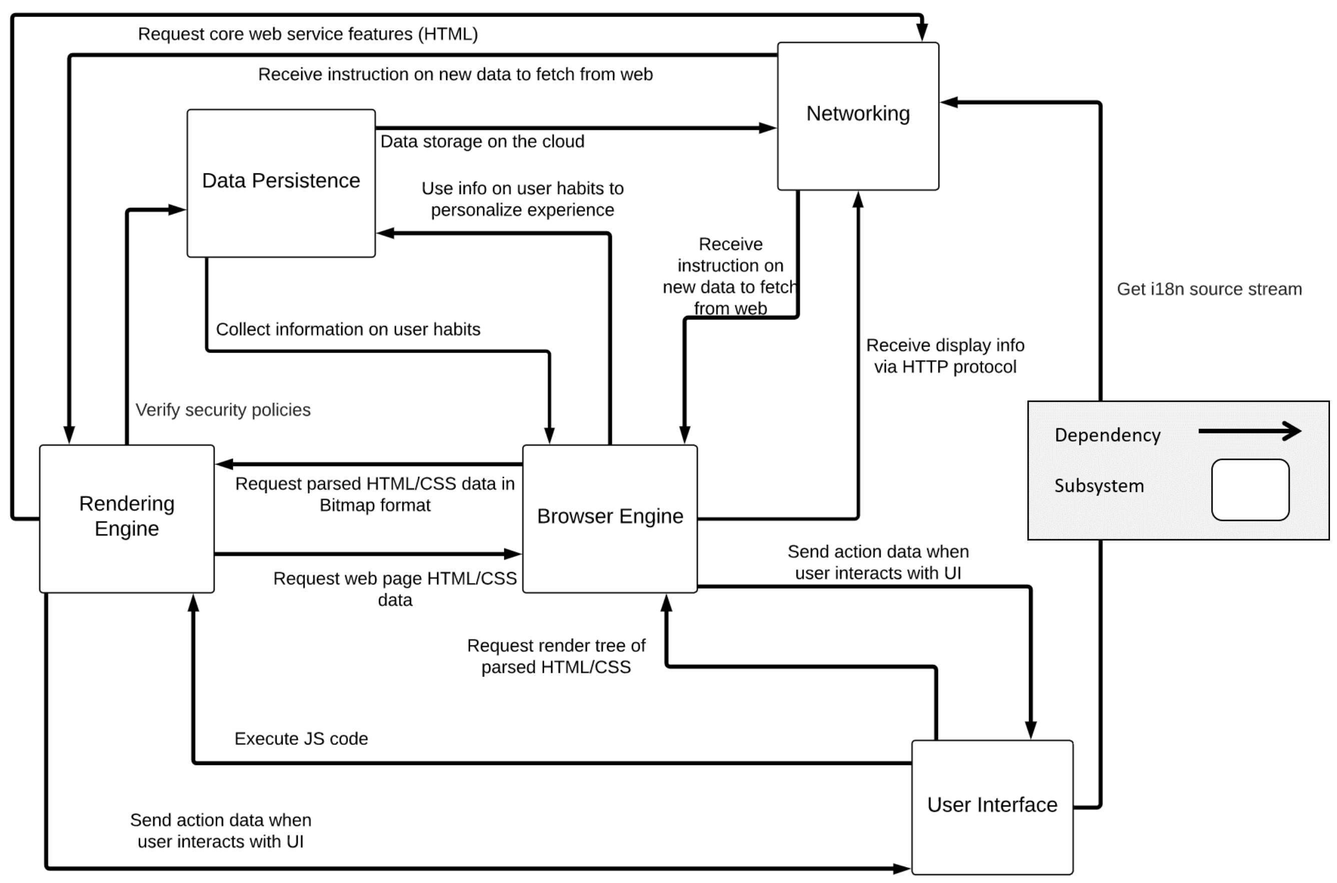

Request core web service features (HTML)
Receive instruction on new data to fetch from web
Networking
Data storage on the cloud
Data Persistence
Use info on user habits to personalize experience
Receive instruction on new data to fetch from web
Collect information on user habits
Get i18n source stream
Receive display info via HTTP protocol
Verify security policies
Dependency
Subsystem
Rendering Engine
Request parsed HTML/CSS data in Bitmap format
Browser Engine
Request web page HTML/CSS data
Send action data when user interacts with UI
Request render tree of parsed HTML/CSS
Execute JS code
User Interface
Send action data when user interacts with UI

Figure 3: Concrete Architecture

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

The Data Persistence subsystem is made up of various components:

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

- **Saved Security Policies:** the rendering engine checks and partitions storage for file system permissions and various other security policies.

The interactions between these components and other subsystems can be seen in Figure 4: Data Persistence Subsystem Diagram.

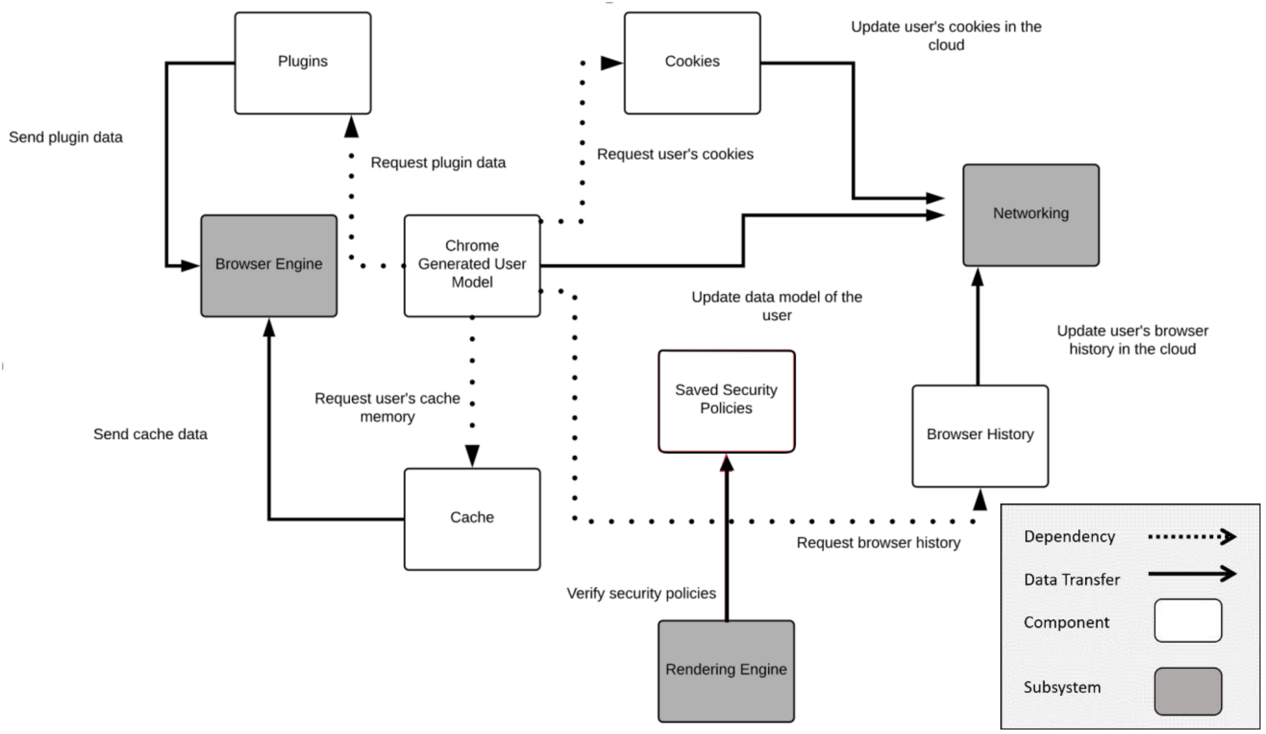


Figure 4: Data Persistence Subsystem Diagram

Concurrency and Data Flow

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

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

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

Render Process

On the Main/UI thread there is one *RenderProcess* object per render process [3]. There is also the *RenderView* object, which communicates with its corresponding *RenderViewHost* in the browser process and the renderer layer [1]. This object represents the contents of one web page in a tab or popup window. This can be seen in Figure 5: Browser-Render Process.

Browser Process

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

The I/O thread has a *channel* which defines methods for communicating across pipes. There is a *ResourceDispatcherHost*, which sends network requests over the internet, and the *RenderProcessHost* object which receives the IPC requests from each renderer [1]. These messages are forwarded to the *ResourceDispatcherHost* [1]. This can be seen in Figure 5: Browser-Render Process.

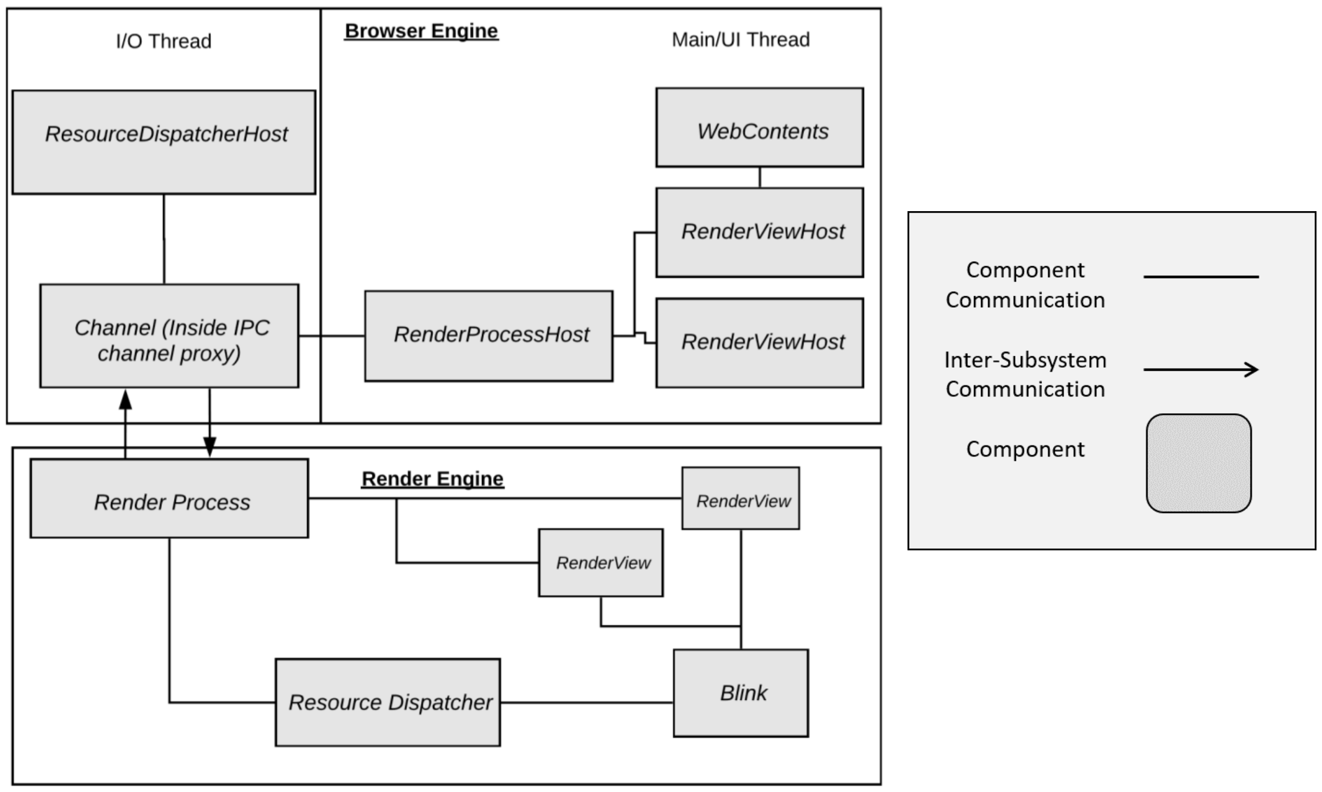


Figure 5: Browser-Render Process

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

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

Reflexion Analysis

There were seven main dependencies that were initially missed by the team. For all seven, the main reason the dependency was overlooked was because it was assumed that the communication between the subsystems would go through the browser engine. However, through investigation of the source code, it was determined that direct connections between certain subsystems are required for functionality and efficiency. This oversight stemmed from the inexperience of the team in browser development and architecture principles, as well as research limitations caused by time restraints and inaccurate information sources during Assignment 1. Each of the new dependencies can be seen in Figure 6: Reflexion Model.

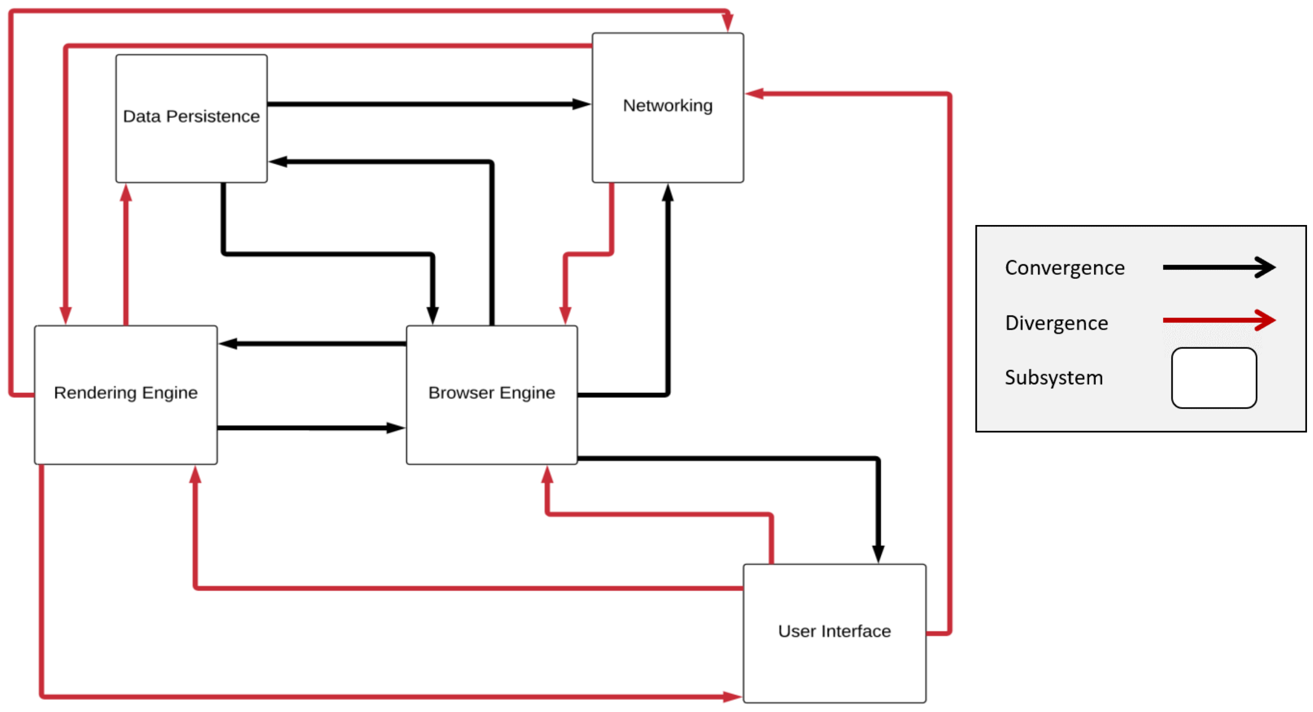


Figure 6: Reflexion Model

UI to Networking

The networking subsystem provides the UI with information on Aura and MUS, which are required by Chromium for window management. The networking subsystem also contains information for the internationalization (i18n) stream, which is used to provide multi-language support in the UI.

Rendering Engine to Data Persistence

This dependency is required for the rendering engine to verify security policies for file system permissions stored in the data persistence storage system. The rendering engine also works to partition data in the storage system during its process.

Browser Engine to UI

It was originally assumed that only a dependency in the opposite direction (UI to browser engine) would be needed. However, a back and forth communication is required to allow the browser engine to adjust backend behavior based on event handling (e.g. for mouse actions) with user interactions in the UI.

Rendering Engine to Networking, Networking to Rendering Engine

These dependencies are needed for the rendering engine to directly receive core web service features (such as HTML and GPU acceleration). The received information can then be passed to the browser engine, which applies changes to the page based on the user's configuration.

Rendering Engine to UI, UI to Rendering Engine

The initial assumption that the communication between these subsystems would go through the browser engine would have been sufficient if all UI content were static. However, direct communication between the rendering engine and the UI is required for control of—and continuous updates to—the dynamic (non-static) content on the page, such as animations, different layers and certain third-party applications.

Developer Implications and Design Tradeoffs

Benefits

The modularity of the individual objects provides building blocks for easy scalability and evolution. New components can be added and tested individually before being implemented into the preexisting system. The design style also makes the system easily testable. The functionality of each object can be tested individually before testing the relationships between objects and components, as well as their combined functionality. The object-oriented structure also works to achieve one of Chrome's primary non-functional requirements of high security. Modularity allows the browser to easily separate functionality and sandbox individual processes.

Issues

In addition to the CPU and memory issues discussed in Assignment 1, the updated concrete architecture can cause new difficulties for the developer team. With so many added dependencies between subsystems, the architecture's complexity—and thus, the development complexity—increases greatly. With numerous components depending directly on the functionality and efficiency of others, it is essential for teams to be aware of how their actions will affect the rest of the system. This also increases the likelihood of bottlenecks within the system, as issues or slowness in one subsystem can cause delays in the entire chain of communication.

External Interfaces

Plugin Architecture

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

In-Process Plugins

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

Out of Process Plugins

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

Storage and Databases

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

Use Cases

The sequence diagram in Figure 7: Successful Login Sequence Diagram describes how the team’s concrete architecture would render the use-case where a user successfully logs into a website using a password.

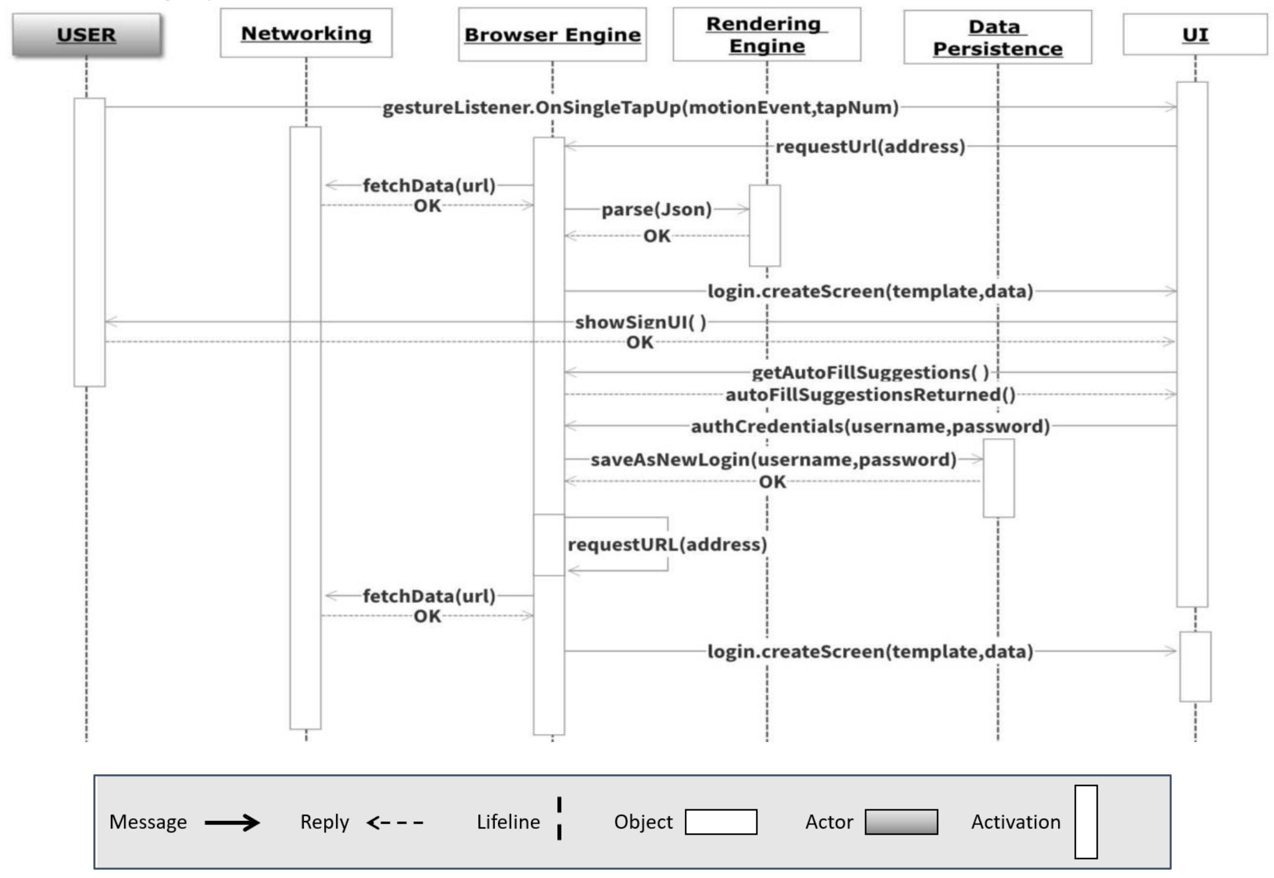
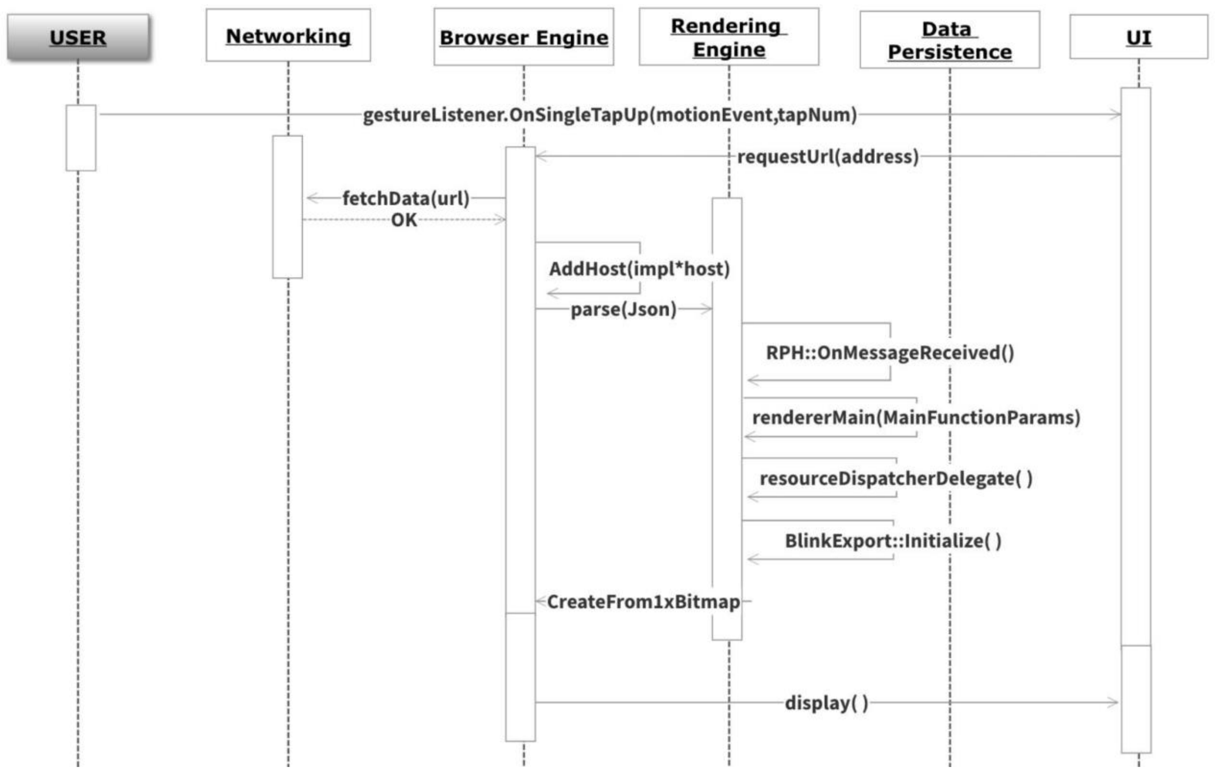

USER
Networking
Browser Engine
Rendering Engine
Data Persistence
UI
gestureListener.OnSingleTapUp(motionEvent,tapNum)
requestUrl(address)
fetchData(url)
OK
parse(Json)
OK
login.createScreen(template,data)
showSignUI( )
OK
getAutoFillSuggestions( )
autoFillSuggestionsReturned()
authCredentials(username,password)
saveAsNewLogin(username,password)
OK
requestURL(address)
fetchData(url)
OK
login.createScreen(template,data)
Message
Reply
Lifeline
Object
Actor
Activation

Figure 7: Successful Login Sequence Diagram

The sequence diagram in Figure 8: Rendering JavaScript Sequence Diagram describes how the team’s concrete architecture would render a page of JavaScript.



Message ⟶	Reply <- - -	Lifeline ¦	Object ▭	Actor ▨	Activation ▯

Figure 8: Rendering JavaScript Sequence Diagram

Data Dictionary

Blink: WebKit-based rendering engine
Browser: represents the browser window, contains multiple instances of WebContents
Cache: space in a computer's hard drive and in RAM memory where the browser saves copies of previously visited web pages. The browser uses the cache like a short-term memory
Channel: defines methods for communicating across pipes
I/O Thread: a thread in the browser process that handles IPC's and network requests, and in the renderer process that just handle IPC's
IPC Channel Proxy: Within the browser, communication with the renderers is done in a separate I/O thread. Messages to and from the views need to be proxied over to the main thread using a Channel Proxy
Main/UI Thread: a thread in the browser process that updates the UI and renderer process that runs most of Blink; responsible for rendering web pages on screen
NPAPI: Netscape Plugin Application Programming Interface, interface that allows browser extensions to be developed. The chrome plugins that have full permissions of the user and is not sandboxed
Parser: compiler or interpreter component that breaks data into smaller elements for easy translation into another language
Renderer/Render host: multi-process embedding layer

RenderProcessHost: initialized on the main thread that creates a new render process for each website
RenderView: responsible for navigational commands, receiving input events and painting web pages
RenderViewHost: receives view-specific messages from RenderProcessHost
RenderWidget: input event handling and painting
RenderWidgetHost: handles the input and painting for RenderWidget
ResourceDispatcherHost: responsible for sending network requests to the internet
Sandboxing: a security mechanism used to run an application in a restricted environment that is cut off from the rest of the computer system
Tab Helpers: separate objects that attach to WebContents
WebContents: embedded to allow multi-process rendering of HTML
WebKit: rendering engine
WebKit Glue: converts WebKit types to Chromium types.
WebKit Port: integrates with platform dependent system services such as resource loading and graphics

Naming Conventions

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

Conclusions

The team learned that they had missed a few important dependencies in their conceptual architecture, which they added in to their concrete architecture. Many of the missed dependencies were the result of assuming communication between subsystems would indirectly go through the browser engine. Though the system architecture was mostly left the same, the added dependencies greatly increased the complexity of the design, adding difficulties for the Chrome development team. More dependencies results in the possibility for more performance issues, as bottlenecks can occur due to slowness and errors in any component of the communication chain.

Moving forward, the team will need to do further investigation into the source code to learn how to properly implement a new facial recognition feature for Assignment 3. This will require the team to have a complete understanding of how the subsystems interact and the functionality of each component. Though the new feature can be built off preexisting components, the team must still take into consideration security and performance concerns.

Lessons Learned

When looking through the source code, the team quickly learned that much of the official documentation provided about the Chromium project is outdated and no longer matches what exists in the code base. The team also realized that, due to the vastness of the source code and time restrictions on the project, it was impossible to investigate all the code at a low level. Much of the research had to be done through assumptions based on file names and explanations provided in README files, rather than looking at the programs themselves. This most likely led to inaccuracies in the designed concrete architecture.

The Understand tool also provided some limitations to the design process. The software had difficulty processing such a large amount of code files at once, leading to disruptions in the investigation and architecture building process. Code had to be loaded into Understand and viewed in segments to avoid a system crash, which greatly slowed down the process and made it difficult to view the entire architecture system at once.

The team also found it difficult to divide work evenly throughout the research and design phases of the assignment. As it was important for each team member to have a general understanding of the entire system and its interactions, tasks could not be easily divided by the separate system components.

References

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

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

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

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

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

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