The Concrete Architecture of Google Chrome

Assignment 2
November 9, 2018 (Fall 2018)

Thick Glitches
Alastair Lewis (15ahl1@queensu.ca)
Andrea Perera-Ortega (15apo@queensu.ca)
Brendan Kolisnik (15bak2@queensu.ca)
Jessica Dassanayake (15jdd1@queensu.ca)
Liam Walsh (15lcw1@queensu.ca)
Tyler Mainguy (16tsm@queensu.ca)

Abstract

A concrete architecture for the Google Chrome web browser was derived through analysis of Chromium, an open-source version of Google Chrome. The concrete architecture was developed through the analysis of source code using Understand, a software intended to help software developers to gain insight into their source code.

Chrome uses an object-oriented architecture style, depicted by its five subsystems and the abstraction between these subsystems. The reflexion analysis technique was applied to the derived architectures to determine what dependencies were justified and what dependencies were unjustified for the unexpected dependencies in the concrete architecture. The browser and networking subsystem were studied in more detail to determine their concrete architectures and any additional dependencies from their subsystems that were not accounted for in the conceptual architecture of the browser and networking subsystems. The concrete architectures have more dependencies than their conceptual counterparts; this is due to their more factual nature since they are derived from source code rather than documentation. While more dependencies exist, Chrome's system is organized in a way that is still optimized for performance. In addition to the derivations of the concrete architecture, use cases are included in this report to further demonstrate how the subsystems interact.

The in-depth study of Chrome on a concrete level allowed for an increased understanding of the system and what can be added to enhance it. A proposed feature is Chrome Safe Mode, which utilizes several subsystems to censor inappropriate content. Further research will be carried out for this feature, however, an overview of the feature is presented in this report.

Introduction and Overview

After previously deriving the conceptual architecture of Chrome, it was determined that the development of a concrete architecture for Chrome was necessary to gain a deeper understanding of the major subsystems and their interactions. Since the concrete architecture is much less abstract than the conceptual architecture, it was necessary to analyze the source code in order to create the most accurate version of the concrete architecture. To do this, the software Understand was used to study Chromium's code. After revisions of both the concrete and conceptual architectures using this tool, research, and use of the reflexion model, a final concrete architecture was derived that consists of five distinct subsystems.

The browser and networking subsystems were determined to be two of the most significant subsystems of Chrome. Following the analysis of the source code, it was decided that the browser is made of five subsystems that interact with each other, while networking contains three. In-depth architectures, both conceptual and concrete, were created to display the interactions between the subsystems within browser and the Chrome subsystems that browser's subsystems are dependent on. This process was also carried out for the networking subsystem.

Both on the highest level (Chrome) and lower levels (e.g. browser and networking), the subsystems are organized in an object-oriented architecture style. The systems are highly optimized for performance which is due to several factors. Firstly, the ability of the system to run processes concurrently is a significant reason for Chrome's success and popularity. Additionally, the system's dependencies are set in a way that allows for high cohesion and low coupling. However, the subsystems are less independent as expected from the creation of the conceptual architecture. While the concrete architecture has more dependencies than the conceptual architecture, the arrangement minimizes coupling as much as possible to increase performance.

By having a greater understanding of the architecture, we were able to come up with a new Chrome feature to be proposed: Chrome Safe Mode. Knowledge of the subsystems allows us to have a good idea about how to implement the feature to enhance Chrome. This proposal will be pursued further in the future to see how we can improve the web browser in its current state.

Concrete Architecture

Derivation Process

The original conceptual architecture was reflected on to see if any revisions could be made. By revising the conceptual architecture, a solid foundation was created to form the concrete architecture of Chrome. By using the Understand software, we were able to analyze the source code and come to conclusions about what the major subsystems are. Specifically, we observed the metrics treemap, organized by lines of code, to determine the subsystems of Chrome. We concluded that more lines of code meant a greater significance for that particular module, however, we did not solely base our conclusions off this. The feature also allowed us to view dependencies between the subsystems. A possible concrete architecture was developed, but ultimately, the final version of the concrete architecture was derived after conducting reflexion analysis to determine which dependencies were justified and which were unjustified.

Revised Conceptual Architecture

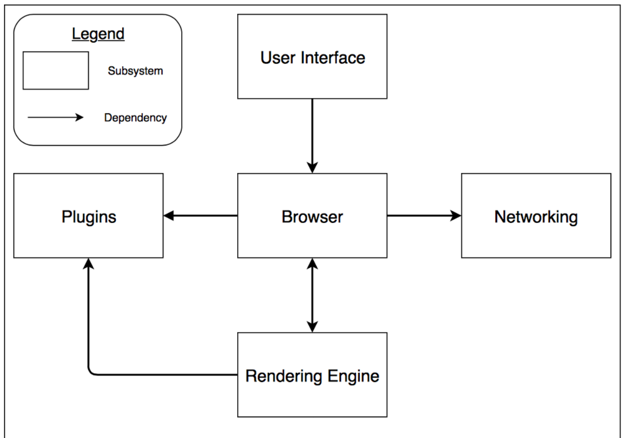


Figure 1: Original conceptual architecture

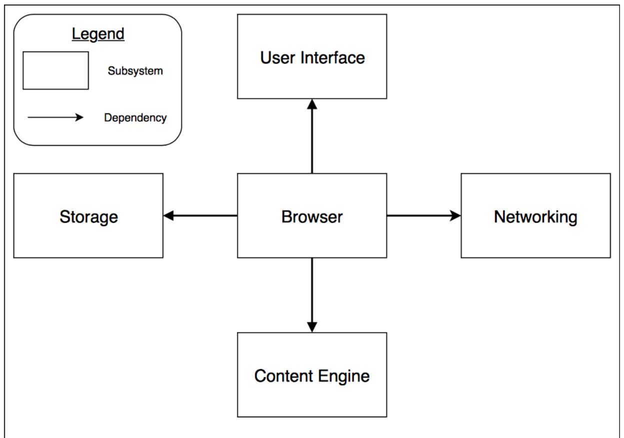


Figure 2: Revised conceptual architecture

With additional research, it was determined that changes should be made to the initial conceptual architecture that was derived. There were several key changes made between the original conceptual architecture (Figure 1) and the revised conceptual architecture (Figure 2). By observing the source code, we noticed that the plugins subsystem could be removed as it was not a key system. Also, we realized a need for the storage subsystem to act as a dedicated module for read and write access to the host machine. The rendering engine was renamed to be the content engine because we found that the rendering of the applications is broken up into two distinct rendering process; the renaming allowed us to be more specific. It was also determined that the browser depends on the UI as opposed to the UI depending on the browser. Furthermore, it was decided that the browser is dependent on the content engine and that the content engine is not dependent on the browser.

Alternative Concrete Architecture

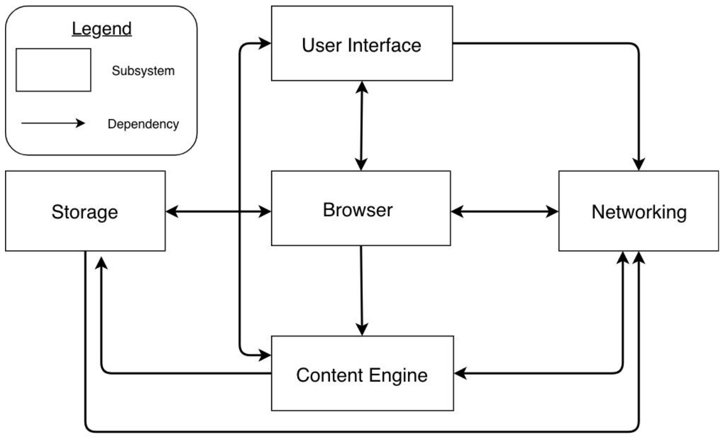


Figure 3: Alternative concrete architecture

The alternative concrete architecture we derived had several key changes from the conceptual architecture developed in assignment one. We changed the render engine to the content engine to reflect the fact that the browser and content engine subsystems both have their own rendering processes. We removed the plugins module as we decided it was not a key aspect of the architecture and instead added a storage subsystem to provide functionality for reading and writing directly to the host computer.

Final Concrete Architecture

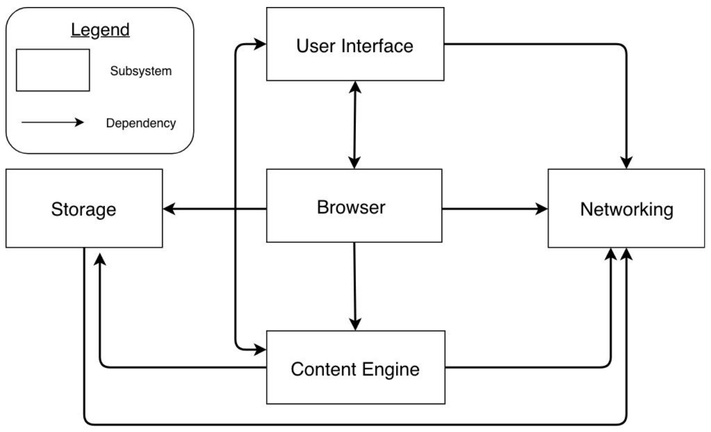


Figure 4: Final concrete architecture

Our final concrete architecture contains all of the same subsystems as the alternative but with some differences in the dependencies between subsystems which will all be discussed in the reflexion analysis section.

Reflexion Analysis

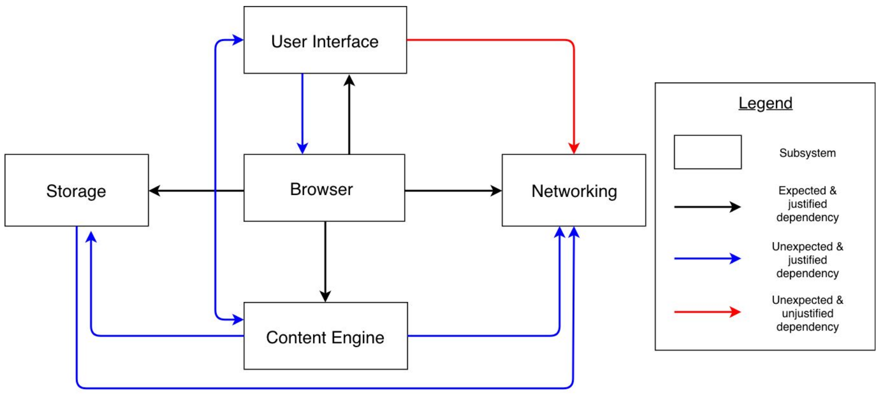

User Interface
Storage
Browser
Networking
Content Engine
Legend
Subsystem
Expected & justified dependency
Unexpected & justified dependency
Unexpected & unjustified dependency

Figure 5: Reflexion analysis of the architecture

Justified/Unjustified	Dependency From	Dependency To	Rationale
Unjustified	UI	Networking	The UI reuses networking's platform-independent code for resolving local path names

Justified	Storage	Networking	Storage is able to directly handle blob downloading/uploading. Blobs are not likely to be malicious. Blobs contain lots of data to route through multiple subsystems, so a direct link is beneficial for efficiency
Justified	UI	Browser	UI needs operating system access to use Apple's framework to fill in the taskbar. Chrome is responsible for drawing anything that isn't tab content
Justified	UI	Content Engine	UI needs to interact with the content engine for features like: Developer Tools. Chrome shell console, inspect element, etc
Justified	Content Engine	Storage	Interacts with storage for file API, blob storage and quota manager
Justified	Content Engine	Network	Requires internet access for handling websockets, hyperlinks, CDN, and disk caching (unconfirmed downloads)
Justified	Content Engine	UI	Every time UI needs something painted/rendered, it communicates directly with the content engine. All event objects in UI are depended on by the content engine

Subsystems

Browser
The browser subsystem is the central subsystem in the application and contains the system's key processes to run the other subsystems. The browser contains its own rendering process, separate from the content engine which is able to render the section of Chrome containing the search bar, bookmarks, tab selector and the settings menu.

User Interface
The user interface is the subsystem that is responsible for displaying rendered content from the browser and content engine and provides the interface for the user to interact with the browser. It is able to detect mouse presses, keystrokes and other activity from the user.

Networking

The networking subsystem is responsible for sending and receiving data from the internet. The networking subsystem is the only one that is able to function independently from all the other subsystems as it has no dependencies.

Content Engine
The content engine is the subsystem within Chrome that is responsible for parsing and rendering all of the content that is received in HTML/JS/CSS format and displaying it to the UI [6].

Storage
The storage subsystem is responsible for providing reading and writing capability directly to the host machines file system. This provides the Chrome with the ability to have data persistence which is useful for a number of features in the browser.

In-Depth Architecture

Conceptual Architecture Browser Subsystem

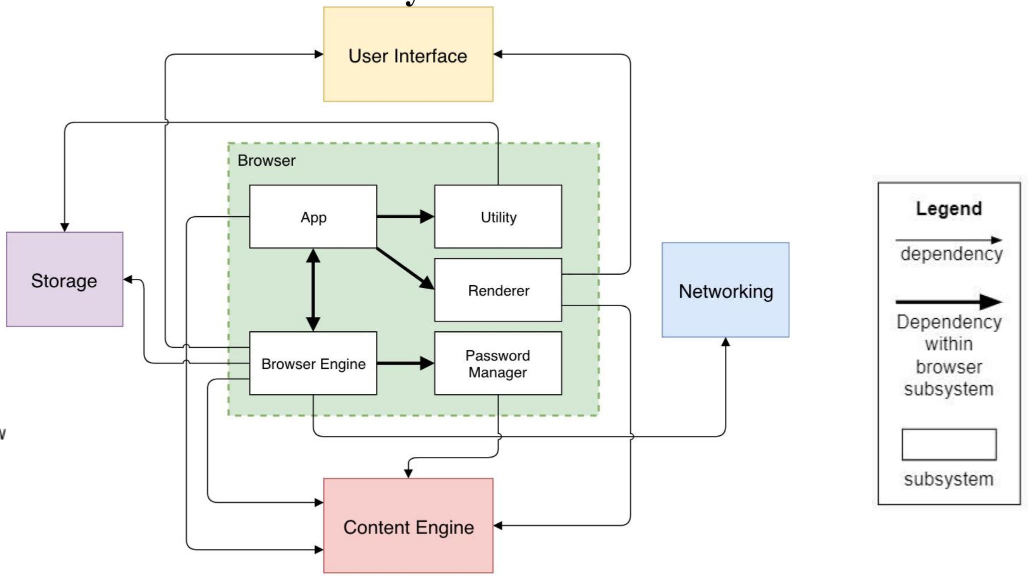


Figure 6:
Conceptual Architecture Brow

Concrete Architecture Browser Subsystem

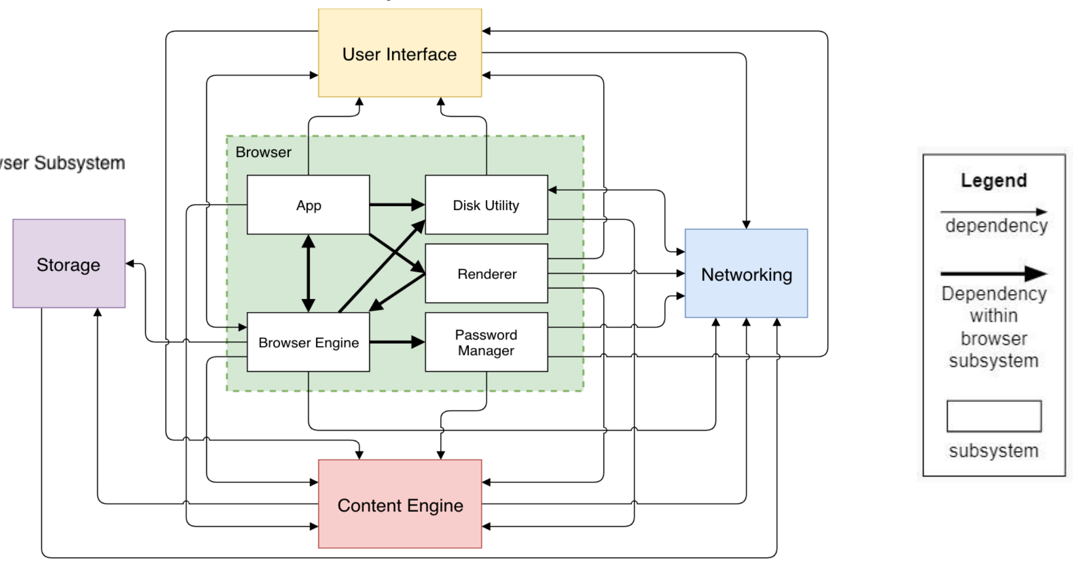


Figure 7:
Concrete Architecture Browser Subsystem

Browser Subsystem Internal Reflexion Analysis

Justified/Unjustified	Dependency From	Dependency To	Rationale
Justified	Browser Engine	Disk Utility	Browser Engine has direct dependency to write cookies and more straight to storage.
Justified	Renderer	Browser Engine	Renderer depends on Browser Engine due to implementation requirements such as drawing and displaying the menu bar for OSX which is handled by the OS.

The conceptual architecture for the browser system is both object-oriented and layered style. The top layer being the app and browser engine systems and the bottom layer being the utility, renderer, and password manager systems [16]. The bottom layer implementations would not need the top layer while the top layer depends on the bottom layer. As with other systems in Chrome, the architecture is object-oriented to increase cohesion and decrease coupling.

The concrete architecture for the browser system somewhat violates the layering design as the Renderer depends on browser engine due to implementation requirements such as displaying the menu bar for OS X which is drawn by the OS. However, this is very limited. The concrete architecture is also object-oriented to decrease coupling and allow implementations of systems to be changed such as when the content engine switched to Blink from Webkit [6].

Conceptual Architecture Networking Subsystem

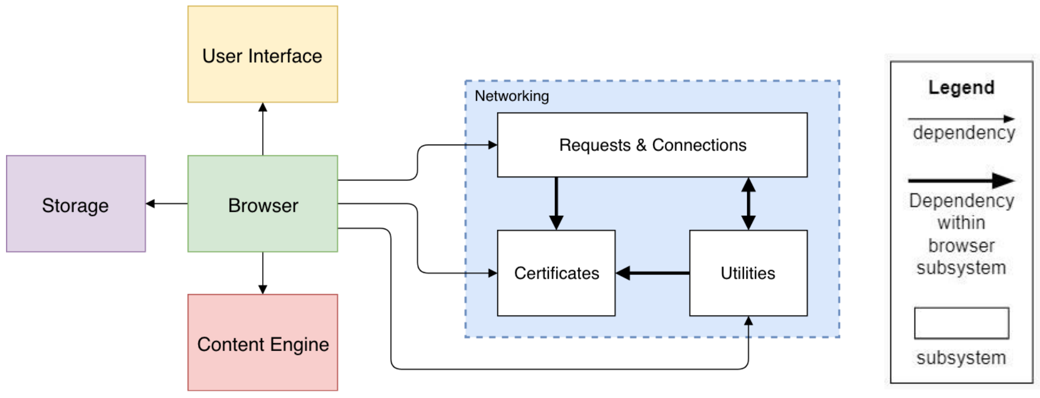


Figure 8: Conceptual Architecture Networking Subsystem

Concrete Architecture Networking Subsystem

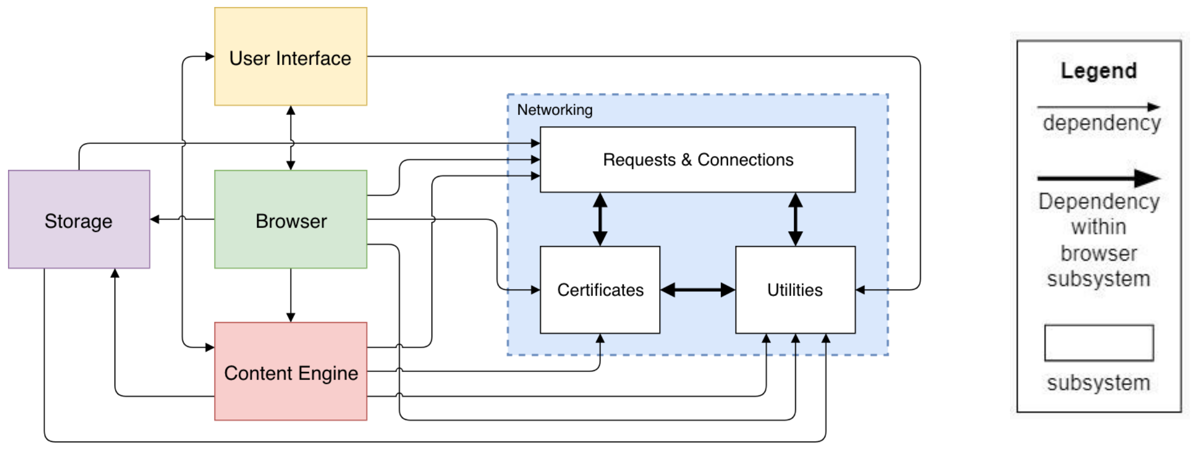


Figure 9: Concrete Architecture Networking Subsystem

Networking Subsystem Internal Reflexion Analysis

Justified/Unjustified	Dependency From	Dependency To	Rationale
Justified	Certificates	Utilities	Certificates uses libraries from the utilities subsystem to help hash, unhash, encrypt, decrypt, etc.
Justified	Certificates	Requests & Connections	SSL sockets are configured by the two subsystems working together. HTTP request headers that require signing are sent to and from the certificates subsystem.

The conceptual architecture for networking is object-oriented. The main subsystem is the Requests & Connections subsystem. The reason this subsystem is not divided into two separate subsystems is that the processes for requests and connections are not very cohesive, and coupled substantially to each other. Most work is done by the Requests & Connections subsystem, with frequent calls to the Utilities subsystem for tasks such as logging, using cookies, disk caching, and more. The Certificates subsystem is used to create certificates for the user's network protocols.

The concrete architecture for networking is even more coupled than it was initially estimated to be in the conceptual architecture. This being said, we believe that the coupling of the networking subsystem is not inherently detrimental to its operability. In fact, the tight coupling helps to increase the speed of operation. As well, since the low-level methods of network communication and internet protocols are standardized, the networking subsystem will almost never need to be changed. This makes its high coupling a non-issue.

Proposed Feature

Our proposed feature for the enhancement of Chrome is “Safe Mode”. Enabling this feature within settings will censor instances of pre-set blacklisted words on a web page or entire websites deemed as inappropriate. In addition, it will also censor and prevent inappropriate content like images from being shown to the user on the screen. It will be possible to activate and deactivate Safe Mode with a password. We feel that this proposed feature is ideal for children as well as people working in professional and/or corporate environments. This proposed feature will affect multiple subsystems, including the browser, UI, storage, and content engine subsystems. Within the browser subsystem lies the password manager subsystem which deals with the functionality of the password whereas the password is actually stored in the storage subsystem. Browser also deals with chrome settings, where Safe Mode can be enabled and disabled. The content engine subsystem is affected because it is where the HTML, CSS, and JS parsing occurs. At the parsing stage, inappropriate words and content will be censored. Unsuitable words will be checked through when the HTML is parsed. Inappropriate images will be checked by analyzing metadata. Metadata that contains inappropriate words or that comes from an inappropriate source will trigger the Safe Mode to censor the content. Finally, UI subsystem is affected because a new button icon will be displayed and implemented into the UI for the browser to toggle on/off.

Concurrency

Concurrency is the ability of a program to run its processes in parallel. Chrome was designed with a multi-process architecture, which allows for concurrency of processes that it runs. This means that each tab and plugin exist in their own processes, independent of one another[13]. The way they are able to implement this is by running a main process in the browser system that communicates and facilitates with the individual render processes through IPC [4][3].

The browser process has an object, called the *RenderProcessHost,* that exists on the main thread of the browser. The main job of the *RenderProcessHost* is to facilitate communication from the child processes to other systems in the architecture, like networking or storage[13]. In the main thread of the content engine, there is a *RenderProcess* object, which corresponds to a render process (for plugins, tabs, etc.)[13]. This process will make network and file system requests, and cross-thread communication requests through the *RenderProcessHost*, which will then facilitate it for them.

The benefits behind having a parent browser process and independent (concurrently running) render processes are efficiency, security, and reliability[13]. Efficiency can be achieved as each render process makes requests through its own IPC with the browser process, which then will make those requests for the render processes[4]. By confirming the requests made by render processes are asynchronous non-blocking, then each request can be made independently of one another. This confirms that no requests have to wait for one another, and slow requests cannot hold up requests made by other processes. This results in a faster and efficient execution. From a security standpoint, separating (or “sandboxing”) processes and making them run requests through a parent process is beneficial. By sandboxing processes from one another, no malicious web pages/plugins can gain access to other processes (and their data). By running all network and file system requests through the parent *RenderProcessHost,* it confirms that no malicious render processes can make requests/run scripts that access the file system of the host machine[13]. Reliability is achieved by removing the single point of failure, which was originally the browser process. By delegating tabs and plugins to their own processes, this confirms that any hung/malicious processes cannot slow down/crash your entire browser.

How the Architecture Supports Future System Changes

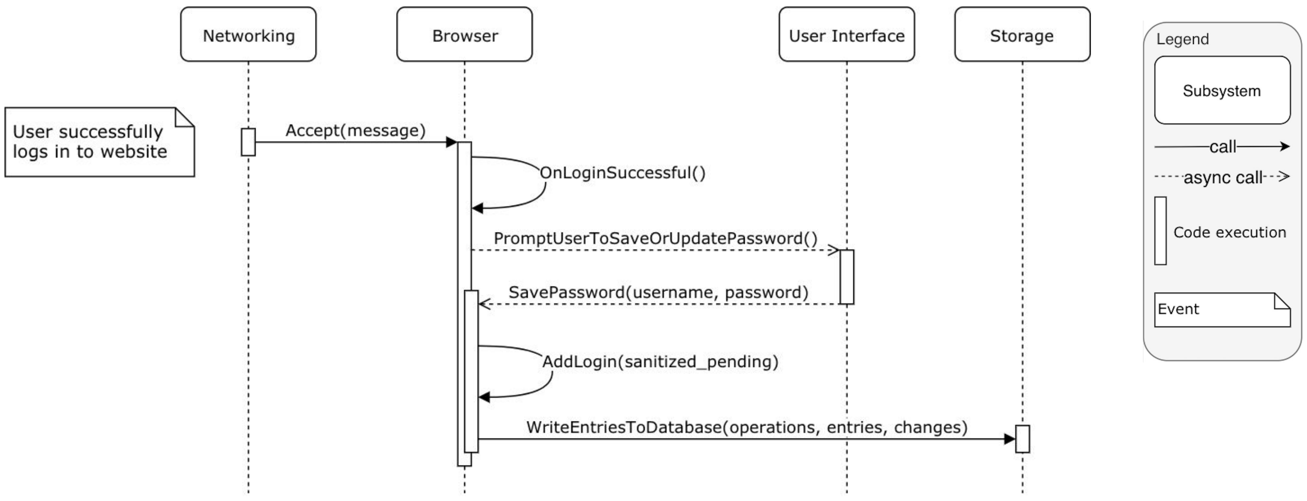
The overall architecture style is object-oriented and some internal systems are layered so one advantage is that implementations of systems can be changed without affecting the overall system. For example, changing the implementation of the content engine would not affect the UI system. The IPC cannot be easily changed as it is tightly coupled into multiple systems including the rendering engine, plugins, and the browser system. This architecture is very extensible. New systems can be added as objects and referenced without changing the implementations or communication systems of other systems. Chromium is the open source code that Chrome is based on which allows the community to contribute new features that can be implemented into production Chrome. Leveraging this strategy, the Chrome team can get the community to implement many of the new features going forward (while monitoring for performance and security). Chromium development also allows new features or web standards to be implemented more quickly in Chrome than in some other browsers helping to keep Chrome on the bleeding edge.

External Interfaces

There are multiple external interfaces that Chrome communicates with: GUI, file system, and network. The GUI allows the browser to process user input events and send visual output to the user by utilizing the UI. The file system allows Chrome to interact with the local files stored on the host machine. Chrome is limited to accessing the files that the operating system has given it permission to read and write to. The browser is able to interface with the network through the networking subsystem, which allows it to send and receive content that is not stored locally.

Use Cases

User Logs into Website and Chrome Saves Username and Password


Networking
Browser
User Interface
Storage
User successfully logs in to website
Accept(message)
OnLoginSuccessful()
PromptUserToSaveOrUpdatePassword()
SavePassword(username, password)
AddLogin(sanitized_pending)
WriteEntriesToDatabase(operations, entries, changes)
Legend
Subsystem
call
async call
Code execution
Event

This sequence diagram describes a user entering a password into a site, confirming the form submission, and saving their password for the site in chrome. Our diagram assumes the user successfully logged into the page.

After the request was successful, the networking subsystem passes a mojo message to the browser subsystem, indicating a successful request[18]. The browser uses this message to run the OnLoginSuccessful() method, which is the start of allowing the user to save their password through chrome[18]. After this is executed, a widget is displayed on-screen (passed from browser to UI), prompting the user to save their password in chrome. After the user clicks “yes”, then the response is sent back to the browser system. After that, the addLogin function is called, with parameter *sanitized_pending*, which deals with SQL special characters, and other potentially malicious text[18]. After the sanitization, the password is then written to the database[18]. There is nothing displayed to the user, but now the password can be accessed in future logins.

Downloading and Displaying a Web Page with JavaScript

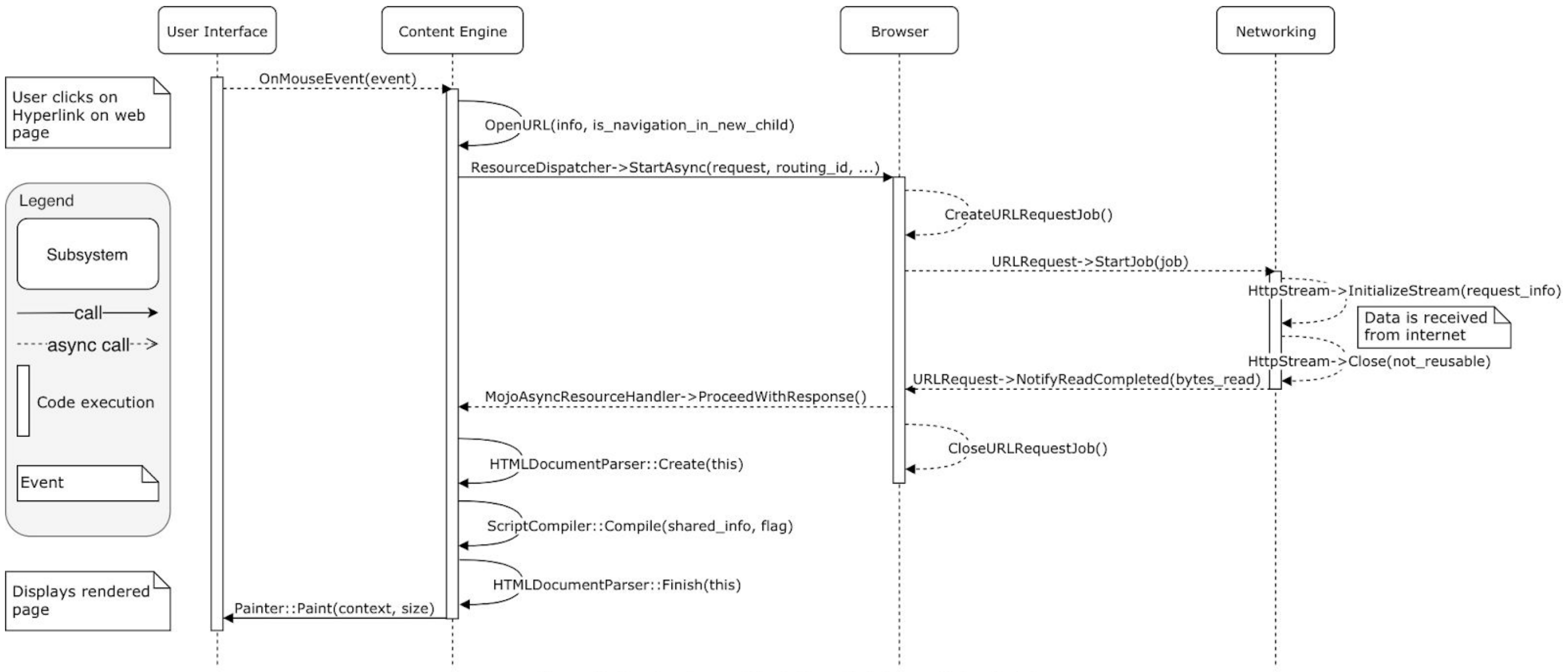

User Interface
Content Engine
Browser
Networking
User clicks on Hyperlink on web page
OnMouseEvent(event)
OpenURL(info, is_navigation_in_new_child)
ResourceDispatcher->StartAsync(request, routing_id, ...)
CreateURLRequestJob()
URLRequest->StartJob(job)
HttpStream->InitializeStream(request_info)
Data is received from internet
HttpStream->Close(not_reusable)
URLRequest->NotifyReadCompleted(bytes_read)
MojoAsyncResourceHandler->ProceedWithResponse()
CloseURLRequestJob()
HTMLDocumentParser::Create(this)
ScriptCompiler::Compile(shared_info, flag)
HTMLDocumentParser::Finish(this)
Painter::Paint(context, size)
Displays rendered page
Legend
Subsystem
call
async call
Code execution
Event

Figure 11. Downloading and Displaying a Web Page with JavaScript

This sequence diagram describes the event of a user rendering a web page with Javascript. This chain of events begins after the user clicks a hyperlink on a web page. This mouse event is sent to the content engine, which is the first step in the rendering process. It runs OpenURL with a boolean value indicating whether or not to make the request in the current tab, or to create a new tab process and continue the request there, and leave the old one as is[18]. Once this is determined, the ResourceDispatcher will start an asynchronous network request, which must be facilitated through the browser component[13]. Once this IPC message is sent to the browser[4], the ResourceDispatcherHost will generate a URLRequest object using the CreateURLRequestJob() method[17]. After the URLRequest object is generated, it is passed via

IPC to the networking system, where StartJob(job) is called. This will trigger the generation of a HttpStream object, which will be responsible for retrieving the request data from the server. This call is asynchronous (as it is a stream of data being read in), and will continue to execute until the server has finished returning data [17]. Once the stream is open (and receiving data), the responses are returned in a URLRequest object, which notifies the ResourceDispatcherHost of the amount of data read, and a pointer to the location of the data[17]. The URLRequestJob will continue to execute until the data has finished being read in. Once the data has been read in, a pointer to the data is sent via IPC to the content engine component[4], so the response can be rendered. From here, the web page HTML is parsed[13]. While the HTML is being parsed, it is watching for any <script> tags, indicating Javascript code. Once a script tag is found, control is handed to the ScriptCompiler object, which will call the Compile method to compile the javascript. Once the HTML has been parsed, the HTMLDocumentParser object is used to run the Finish method, indicating the page has been rendered fully[13]. It will then send the painted page through IPC to the user interface component. The UI will then display the rendered web page.

Team Issues

There are a multitude of issues that arise for development teams when working on a large scale application such as Google Chrome. Unjustified and/or unclear dependencies make it particularly difficult for developers working across systems to understand certain aspects of the code that they were not involved with during implementation. Since Chrome implements Chromium's source code and anyone is able to contribute (open source), code contributed by random individuals not involved in any Chrome development team can be hard to trace and understand, especially if their contribution is a hack solution. Additionally, more dependencies in the concrete architecture versus the conceptual architecture leads to suboptimal coupling and the necessity for teams working across different subsystems to communicate efficiently. Finally, the migration of Chrome's proprietary interprocess communication system (IPC) to Mojo improved inter- and intra-process communication handling which made concurrently running systems easier for development teams to work with. However, this migration and change was difficult for developers to adapt to since it affected all subsystems and lead to a lot of time and money spent, as well as a lot of frustration.

Limitations and Lessons Learned

Throughout the course of this assignment, we encountered several limitations that required our team to work together and overcome them. Our most prevalent issue during this project was the overwhelming nature of the source code of Chrome. A combination of things made the source code difficult from the start. These include the massive number of lines of code and the fact that the source code is written in C++, a programming language no one in our group was significantly experienced with. Our limited knowledge of this language combined with the lack of comments in the source code made it challenging to understand what each function was doing. In addition, it is also easy to lose track of scope when tracing through the code and we found it difficult knowing when to stop. Another challenge we encountered was the steep learning curve of the program Understand. Before we sought out help from the TAs, the program would crash on us often, leaving us frustrated with the software, and unaware of our next steps. Finally, due to the increased amount of information we needed to research and the more technical nature of it, our group was required to organize meetups far more often than the last assignment. Due to the increased number of meetings coinciding with weeks filled with midterms, group members were under significantly more stress compared to the previous assignment.

Fortunately, challenges are often accompanied with valuable lessons learned. Our unfamiliarity of C++ syntax turned into a valuable lesson and resulted in our group learning the basics of a new programming language. Our frustration with Understand forced us to reach out for help, and from this, we learned two lessons. First, after we spoke with TAs, we gained an understanding of this new software and secondly, we learned that it is important and okay to ask for help when one needs it. Finally, taking on a large group project like this developed our group's organizational skills as a whole.

Conclusion

In summary, the derived concrete architecture of Google Chrome was formed using the conceptual architecture, source code analysis, and reflexion model. This new architecture model presented several unexpected dependencies, which demonstrated that the subsystems are less dependent than what we originally thought when deriving the conceptual architecture. As a result of this, coupling was higher than expected. However, the organization of the subsystems in its object-oriented style is set in a way to minimize coupling as much as possible. These observations carried over to the in-depth conceptual architectures of the subsystems, demonstrated by our analysis of the browser and networking subsystem.

A new feature that we discussed was Chrome Safe Mode, which we believe would be widely accepted by many users for both personal use (filtering for children) and business use (filtering content to be work-appropriate). In order to keep Chrome competitive in its market, new features are always being added. This is possible as a result of the object-oriented architecture style with some layering. This allows for subsystem implementations to be modified without compromising the entire system.

Data Dictionary/Naming Conventions

Bitmap: A mapped output generated by the rendering engine to be displayed in the user interface graphically
Blob: Acronym for "Binary Large Object", a large container of binary data which is passed through the system and is generally not malicious due to the object not being executable.
Browser: The main module in the program that controls data exchange across modules in the application. It is the only module in the architecture that directly interacts with the operating system in order to have functionality for data persistence and resource allocation.
Cohesion: The level of separation of functionality between modules in the software. A highly cohesive system will have modules that perform very specific tasks tailored to their attributes.
Concurrency: Multiple software processes running at the same time by utilizing multiple processing cores.

Coupling: The number of dependencies your program has in between its various modules. It is optimal to have low coupling in your system so that if one module malfunctions there is minimal effect on the other modules in the architecture.
CSS: (Cascading Style Sheets) Lightweight coding language that directly interacts with HTML to allow web pages to have custom appearances and layouts.
Data Persistence: The ability of an application to access the host machines file system in order to read and write to files. It allows for the storage of various types of data such as history, bookmarks, and passwords so that when the application is closed and reopened it still has data stored from the previous browsing session.
GET Request: A request that is sent out into the internet with relevant data for the query stored in the request header.
HTML: (Hypertext Markup Language) The format in which web pages are written in. Uses text with 'tags' surrounding them to describe how they should look and appear on the page.
HTTP: (Hypertext Transfer Protocol) The protocol used by the internet to send HTML across the web from servers to clients.
IPC: (Interprocess communication) Set of programming interfaces that allow for communication and coordination between software processes running concurrently.
JavaScript: An event-driven programming language that shares similar syntax to Java that allows web pages to be interactive and have features that would otherwise be impossible with only HTML.
Modularity: The process of subdividing software into individual components. It results in more understandable code and allows for changes to individual modules without having it affect all others.
Multi-Processing: Software that is designed and optimized to run on computers with multiple processing cores.
Object-Oriented Architecture: Architecture style with multiple, single-purpose 'Objects' that have interfaces to interact with other objects in the system. Objects are not concerned with the implementation of the other objects in the system, only with the data getting passed to them.
POST Request: A request that is sent out into the internet with relevant data for the query stored in the request body as opposed to the header
Reflexion Analysis: The process of examining, understanding and explaining unexpected dependencies in between subsystems of a concrete architecture with the objective of deciding whether or not to keep them in the architecture
Sandbox: The programming practice that segregates a certain set of functionality to a restricted amount of resources, file access, and operating system interactions on the computer in order to increase security. This way, it the sandboxed portion of the application crashes or is compromised, it will have no serious affect on the rest of the system.
UI: (User Interface) The graphical representation of the application that interacts with the user to show/get input and output from them.
URL: (Uniform Resource Locator) A string of text that identifies a specific web page on the world wide web. These are used by the web browser to retrieve specific pages requested by the user.

References

[1] "Blink." *The Chromium Projects*, www.chromium.org/blink.

[2] "Desktop Browser Market Share Worldwide." *StatCounter Global Stats*, gs.statcounter.com/browser-market-share/desktop/worldwide/#monthly-201801-201810-bar.

[3] "Google Chrome." Wikipedia, Wikimedia Foundation, 16 Oct. 2018, en.wikipedia.org/wiki/Google_Chrome.

[4] "Inter-Process Communication (IPC)." *The Chromium Projects*, www.chromium.org/developers/design-documents/inter-process-communication.

[5] "Multi-Process Architecture." *The Chromium Projects*, www.chromium.org/developers/design-documents/multi-process-architecture.

[6] "The Security Architecture of the Chromium Browser". Barth, A., Jackson, C., Reis, C., & Google Chrome Team, 16 Oct, 2018. http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf

[7] "Multi-Process Architecture" *Chromium Blog*, https://blog.chromium.org/2008/09/multi-process-architecture.html

[8] "Network Stack", *The Chromium Projects,* https://dev.chromium.org/developers/design-documents/network-stack

[9] "Disk Cache", *The Chromium Projects* https://dev.chromium.org/developers/design-documents/network-stack/disk-cache

[10] "HTTP Cache", *The Chromium Projects* https://dev.chromium.org/developers/design-documents/network-stack/http-cache

[11] "CookieMonster", *The Chromium Projects* https://dev.chromium.org/developers/design-documents/network-stack/cookiemonster

[12] "Why is Chrome Using So Much RAM? (And How to Fix it Right Now). Albright, Dann, 17 June, 2017 https://lifehacker.com/why-chrome-uses-so-much-freaking-ram-1702537477

[13] "Explore the Magic Behind Google Chrome". Zeng, Dico, 21 Mar, 2018 https://medium.com/@zicodeng/explore-the-magic-behind-google-chrome-c3563dbd2739

[14] "Which WebKit Revision Is Blink Forking from?" Google Groups, Google, 18 Apr. 2013, https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/J41PSKuMan0/gD5xcqicqP8J.

[15] "Understanding How the Chrome V8 Engine Translates JavaScript into Machine Code." FreeCodeCamp.org, FreeCodeCamp.org, 20 Dec. 2017, https://medium.freecodecamp.org/understanding-the-core-of-nodejs-the-powerful-chrome-v8-engine-79e7eb8af964.

[16] Kosaka, Mariko. "Inside Look at Modern Web Browser (Part 1) | Web | Google Developers." Google, Google, Sept. 2018, https://developers.google.com/web/updates/2018/09/inside-browser-part1.

[17] "Life of a URLRequest"
https://chromium.googlesource.com/chromium/src/+/master/net/docs/life-of-a-url-request.md?fbclid=IwAR0k_r8KKCXzXAaRRV_C7xZASVR3cX-IHmh0MK4wyHLvxwea5wj25uTOIlI

[18] "Code Search", Google Chrome. https://cs.chromium.org/