Assignment 1
Conceptual Architecture of Google Chrome

arCHROMEtects
Jesse Burstyn (5jb13 - 529 4477)
Katricia Barleta (7mkb - 563 4891)
Lukas Berk (7lsb - 574 5228)
P Bennett Cole (7pbc - 561 5386)
Tom Franzon (5tagf - 525 5746)
# Table of Contents

Table of Contents .................................................................................................................1
Abstract .................................................................................................................................2
Introduction and Overview ....................................................................................................3
Conceptual Architecture ........................................................................................................4
   Derivation Process ................................................................................................................4
   Final Result ..........................................................................................................................4
Subsystem Functionalities .....................................................................................................6
   User Interface .....................................................................................................................6
   Browser ...............................................................................................................................6
   Data Persistence ..................................................................................................................7
   Networking (Network Stack) ...............................................................................................7
   JavaScript Interpreter (V8) ...............................................................................................7
   Rendering Engine (WebKit) .................................................................................................8
   Display Backend (views, WTL, Skia & GDI) ................................................................. 8
   XML Parser (libXML & libXSLT) .....................................................................................9
   Plugins ...............................................................................................................................10
      Plugin Modifiability .......................................................................................................10
Concurrency and Threading ..................................................................................................10
Project Development ..........................................................................................................11
Use Cases .............................................................................................................................12
   Fetching a Basic Web Page (XHTML & CSS) Which is Not in Cache .........................12
   Stretching a Browser Window .........................................................................................13
   Fetching a Web Page (XHTML & CSS) with JavaScript and a JavaScript Cookie ..........14
Conclusions ..........................................................................................................................15
Limitations and Lessons Learned .......................................................................................15
Glossary ...............................................................................................................................16
   Chrome Components & Terminology ..........................................................................16
   Web Terminology ..........................................................................................................16
References .............................................................................................................................17
Abstract

In this report we present an analysis of the Google Chrome web browser. The goal of this project was to analyze the available documentation on the Chromium project and determine a conceptual architecture for the system. We determined that the overall architecture style was semi-strict layering, while several subsystems exhibited an object-oriented style at a lower level. These subsystems and their functionalities are described in further detail. Also included are three use cases for typical web browsing functions, following the flow of interaction and dependency of the system. Overall, this project serves as a stepping-stone for subsequently determining the concrete architecture of Google Chrome.
Introduction and Overview

Google Chrome is an open-sourced web browser designed for the modern day Internet user, and the applications that dominate the web. Tasks associated with the average web browser include surfing the Internet for web pages, sharing or sending information to others, or storing information online. However, as the Internet evolves to become more dynamic and interactive, the benchmark for the functionality of the average web browser has stepped up. The web browser not only has to load static HTML web pages, but also handle other things like videos and applications that rely heavily on things like JavaScript and Flash. Consequently, Google has taken this heavy demand and built a browser whose functionality thrives upon the following quality attributes: simplicity, performance, stability, and security.

Google Chrome offers a simple user interface, which alludes to the principle that is best described in the statement, “Simple interface, powerful core” (GoogleDevelopers, June 2009). Simplicity allows users to get to where they want and what they want in an efficient and natural order. Therefore, it facilitates an experience unburdened by the underlying complexities of the architecture, allowing the focus to remain on the user’s tasks and goals.

The multi-process architecture of Google Chrome differentiates it from other browsers. Each tab has its own process, which runs independently from the browser. This allows one tab process to dedicate itself to a single web-application, thereby increasing browser performance. The multi-process architecture is a key facet of Chrome that caters to the rising complexities of the application-dominated web. For instance, when a number of memory-demanding applications share a heap, performance issues arise. Thus, applications that have their own processes and memory pools will out perform those that do not.

The multi-process architecture also increases the stability of the browser, as it provides insulation. In the case that one process encounters a bug and crashes, the browser itself and the other applications running in parallel are preserved. Functionally, this is an improvement over other browsers, as highly valuable user information in other tabs will be preserved.

The Chrome developers have focused intently on security. Browsers are continuously faced with the challenges of attackers that take advantage of system vulnerabilities. Chrome's architecture addresses security with the sandboxing principle, which takes effect of four Windows security mechanisms. These mechanisms reduce the privileges of the rendering engine instances (individual tabs) by restricting their access to the operating system.
Conceptual Architecture

Derivation Process

In our pursuit to derive Chrome's conceptual architecture, we first researched sources that were aimed at explaining the main goals behind Chrome to get a basic understanding of the reasons for its development. Sources such as the Google Chrome book, the YouTube video entitled, "The Story Behind Chrome," the Wikipedia page, and numerous blogs and articles were helpful in determining that Chrome was a new type of browser - one aimed at catering to today’s application-dominated web in contrast to the days when web browsing entailed viewing basic HTML pages. We found that the developers of Chrome set out to create a browser that excelled in the following quality attributes: performance, usability, stability, and security.

After we had a solid understanding of the principles behind Chrome, we found that the Chromium Developer Documentation served as the best source for determining the conceptual architecture, in particular the Design Documentation. Being open-source, the developers at Google have created documentation to help third party developers add functionalities to improve Chrome. This type of documentation was more low-level and thus made it harder to determine the conceptual architecture, but it was still the best source we could find. It will be more helpful in determining the concrete architecture and in adding a new feature to Chrome in the next two parts of the project.

Before we came to our final result, we had some initial ideas about what architectures were present in Chrome. At first, we thought that Chrome may have been fully object oriented, given the high level of data abstraction and extensive use of API's. We realized, however, that at the highest level this was not the case. It also came to mind that there may have been some implicit invocation architecture present because of event handling in the browser UI and in page content. Further research on Inter-Process Communication, however, led us to believe otherwise.

Final Result

The reference architecture of a web browser maps comfortably to Google Chrome's subsystems, as it contains all the components in the web browser domain that are portrayed in Figure 1 through red dash lines. However, there are significant differences within the dependencies of Chrome and that of the reference architecture. The main difference we found was the integration between the browser engine and the data persistence into the browser subsystem. Moreover, the network and rendering engine do not directly interact with each other, but communicate from the browser. Therefore, after further research and after mapping out the subsystems interactions, we have concluded that the overall conceptual architectural is indeed a semi-strict layering style at the highest level.

However, when one microscopes into the implementation details of the system at lower levels, one would see object-orientated behavior due to the coupling between the interacting agents. An
example would be other uses of JavaScript such as auto proxy configuration which would couple the browser with the V8 JavaScript Engine.

Figure 1 - Our derived conceptual architecture of Google Chrome

In general, a layered architecture is categorized as applications with distinct classes of services that can be organized in a hierarchy (Hassan, 2009). Thus, a subsystem at the bottom of the hierarchy ideally does not speak with non-adjacent levels in its own hierarchy. An advantage that arises from this is that internal enhancement of subsystems becomes a much easier task. Moreover, a single layer also lacks the knowledge of the dependencies of the other higher-level layers and their interactions. Hence, this is all depicted in our architectural diagram for Chrome because of the obvious distinctions between the browser kernel, rendering engine, and the other levels. Furthermore, the main part of web browser involves the UI, browser and renderer hierarchy, and is a good example of the fluid top-down dependency throughout the entire system.

In addition, Chrome's subsystems have reasonable cohesion, since their internals do not directly depend on other levels, and communication between levels are done through intermediate threads like the Inter-Process Communicators between the browser and the renderers. Therefore, restricted visibility between connectors, or procedural calls is preserved. Lastly, as a result of this cohesion, the layers are also reusable. Therefore, different implementations with the same interface of a particular layer can be used interchangeably. For instance, another JavaScript interpreter like SpiderMonkey could replace V8 without drastically impacting the rendering engine.
Subsystem Functionalities

With the high level architecture of Chrome defined, we are now ready to delve into the various components and their functionalities, while underlining the ways they interact with one another.

User Interface

The user interface is the subsystem that the user explicitly sees. It is the layer that allows the user to access the functionality of the browser in a method that ideally should be understandable and easy to use. At the highest level, the UI is made up of client and non-client areas. Non-client areas consist of the areas of the UI that the user does not typically interact with, usually consisting of the title bar and window borders.

Conversely, elements that are interactive, such as windows, buttons, menus, confirmation dialogs, and text fields fall under client areas. When a user interacts with a client area, a message is passed to the browser engine to handle the event. For example, clicking on the 'Back' button would request the browser engine to fetch the last visited page, most likely residing in the network cache.

Chrome blurs the line of the client/non-client divide with their custom UI. For example, the unique feature of presenting tab widgets within the toolbar required a new framework to generate these types of widgets, which also generated some controversy within the technology community concerning the design choice.

Browser

In terms of managing the different subsystems, the browser is the most central component of Chrome. The browser is responsible for the following list of tasks:

- Spawning new tabs
- Communicating with the network
- Handling user input
- Window management
- Location bar
- Disk cache
- Cookie database
- History database
- Password database

In a typical browser architecture, the rendering engine possesses sufficient access to the operating system, in particular to the network and the display backend; however, in Chrome the rendering engine must communicate through the browser to obtain any service beyond rendering-specific services.

From a security standpoint, the browser maintains the same system-level privileges as the user and is "outside" of the sandbox. In creating a rendering engine instance, the browser also defines a security policy for that instance, which defines the privileges granted to that rendering engine.
Data Persistence

Data persistence defines any data that will outlive the execution of Chrome and includes the following components: cookie database, history database and password database. Data persistence is achieved through browser--file system communication in most cases. A slight deviation from this model includes file uploads and downloads, cases when the rendering engine will need access to the file system. When uploading a file, the browser process delegates which files are accessible to the rendering engine (determined by the user in a file picker dialog) and gives the rendering engine instance access to these files. Once the rendering engine instance has been closed, however, these permitted files are now determined inaccessible, preventing another tab from gaining access to these files. When downloading files, the rendering engine tells the browser to save them in a designated directory, which helps to protect the integrity of the file system.

Networking (Network Stack)

The network stack has very few dependencies on the rest of Chrome and sits outside of the browser. Stated goals of the network stack are to remain portable (which they have accomplished by having only three dependencies on the rest of Chrome), and independent of both WinInet and WinHTTP. Currently the network stack has been completely re-written twice by Google.

The network stack handles all Universal Resource Locator (URL) requests by the browser, fetches the resources from the network, and also requests caches of results for further possible use. This subsystem only depends on three other parts of Chrome: the browser, V8, and the GoogleURL library. This low level dependency is what would allow the networking subsystem to be ported easily to other operating systems (other than Microsoft Windows). In regards to threading, the network stack runs primarily on the I/O thread and is generally single threaded (other than top level URL Request class).

JavaScript Interpreter (V8)

The JavaScript Interpreter in Chrome is V8, which was developed by a Google team in Denmark and is currently maintained as an open source project. There are three main features that distinguish V8 from any other JavaScript interpreter out there today. The first of these features is hidden classes. Normally when you have two or more separate yet similar objects in JavaScript, they do not share any common structures. Hidden classes act as extra objects which declare the number of common properties between these otherwise distinct objects (see Figure 2 for an example). The second unique feature of V8 is that it generates machine code instead of using an interpreter. The lack of interpreter means there is no generic lookup for a property. Instead, the machine code fetches the property directly out of the hidden object. This was done primarily to increase the speed of JavaScript execution while remaining platform independent. The final feature specific to V8 is precise garbage collection. This process is designed to be very lightweight in order to provide a smooth experience when JavaScript is interacting with several applications within the browser.
Figure 2 - Hidden classes describe common properties between otherwise totally separate objects

Rendering Engine (WebKit)

WebKit is an open source project to layout web pages, taken from Apple. WebKit consists of three components: WebCore, JavaScriptCore and WebKit (an API layer around the two former); however, only the WebCore component is used in Chrome, as all JavaScript is run by Chrome’s V8 JavaScript interpreter and the Chrome team created their own “glue” to act as an API. WebCore offers rendering engine services to Chrome, handling web components such as CSS, DOM, HTML, and XHTML. An API referred to as the WebKit glue enables communication to the renderer and allows the Chromium application to use its own coding styles, coding layouts, and naming systems.

Display Backend (views, WTL, Skia & GDI)

The display backend is responsible for three main tasks: widget creation, graphics rendering, and font rendering. Within our conceptual architecture we grouped all of these under one subsystem. Reading through the Chromium documentation revealed that these are handled by separate components with different implications based on who uses them. The latter tasks - graphics and font rendering - are essential for displaying web pages.

Four components make up the display backend - views, Windows Template Library, Skia, and GDI. Views is Chrome's high level widget framework for custom UI. Windows Template Library is an open source Microsoft API for GUI creation in Win32 applications. GDI is a deprecated Windows graphics render, although Chrome uses it only for its Windows-native text rendering. Skia is Google's in-house graphics renderer, which provides high performance graphics rendering for everything but text (and in some select cases, that as well).

When WebKit parses the HTML of a page, it will call upon components of the display backend to render the content of the page so it can generate a bitmap for display. Here, Skia handles the painting of images and creation of vector graphics from a site. Furthermore, the majority of
content in many web pages is text. GDI takes the unformatted plain text passed from WebKit and renders it in the appropriate font, style, and size onto the bitmap.

The user interface also depends on the display backend to create and manage its UI elements, or widgets. The views framework calls the Windows Template Library to generate basic widgets, such as text fields, check boxes, radio buttons, etc. This established toolkit provides high level access to Win32 libraries for creating Windows default style user interfaces. For Chrome's unique look, the views framework creates all complex widgets that should not have a platform-specific look or implementation. Examples of these elements include windows, menus, and buttons. This also includes one of Chrome's more prominent UI features, displaying tab widgets within the toolbar (traditionally a non-client area), a choice that is not natively supported by any of the planned platforms for the browser. When views creates these widgets, it requests the services of Skia to generate the vector graphics and GDI for the text to be painted onto the element itself.

![Figure 3 - Zoomed in view of the display backend's architecture](image)

**XML Parser (libXML & libXSLT)**

As time progresses, XML and its associated technologies are becoming more and more of an integral part of the web browsing experience. The development team decided to go with more third party, open source code for handling XML in their browser. libXML is an open source package originally developed for use in the Gnome project, and was coincidentally written in C. Although the package itself is not cross-platform, it is easily ported and there are numerous duplicable ports of it already existing. These features, as well as the fact that it is small, lightweight, reliable and still a very active project, made selecting the package a no-brainer for the team.
One thing that the libXML package itself does not support, however, is XSLT. To add this functionality to the browser, the team went with another package from the same team that developed libXML: libXSLT. libXSLT is based on libXML and therefore ties in well with it.

What Chrome does not have is an RSS or Atom feed reader. It also does not display plain XML files in as friendly of a format as some of its competitors do. However, it does support the use of XML and the majority of its associated technologies (for example XInclude, XPath, XPointer, XSL & XSLT).

Plugins

Plugins provide a specific function to the browser that is not already contained therein; for example, the Adobe Reader plugin allows PDF documents to be viewed inside Chrome and the QuickTime plugin allows videos to be viewed inside a Chrome tab.

Chrome places plugins outside both the browser and the rendering engine. By keeping plugins outside the browser, more stability is afforded to the browser because plugins traditionally represent a large portion of browser crashes. Due to Chrome’s multi-process architecture, the plugins cannot reside in the rendering engine because plugin developers expect that there will be at most one instance of a plugin for the entire browser. For example, if there were two tabs running QuickTime videos, then there would be two plugin instances. Another reason plugins cannot reside in the rendering engine is because they must have direct access to the operating system, and the sandboxing of the rendering engine would prevent this.

Chrome creates separate processes for each plugin that is in use (just as the browser and the renderers are run in separate processes), each of which contains an interface for communicating with the browser and the rendering engine. When a web page requiring plugins is visited, a new process is created, and when all tabs that use the plugin are closed, that process is destroyed.

Plugin Modifiability

Future plugin implementations may take advantage of Chrome's sandboxing mechanisms. Currently, plugins possess the same system privileges as the user because they are contained in neither the browser nor the rendering engine. Plugin developers could write new plugins that operate within the sandbox, preventing attackers who exploit plugin vulnerabilities from installing malware on the user’s machine. Chrome makes testing sandboxed plugins easy by using the --safe-plugins flag when running Chrome from the command line.

Concurrency and Threading

One of the main features of Chrome is that it employs a multi-process architecture. The main obstacle to overcome when attempting to implement such an architecture is creating an efficient way for all of the processes to communicate. The development team explored using a few
existing options (like COM), but eventually decided on using named pipes. However, piping is considerably different on different systems, so to implement this they set out on creating something that has become known as the IPC (Inter-Process Communication) channel.

The threading model is, at its implemented level, quite complex. Keeping it simple, what basically happens is that there is one main process handling user interaction, the browser, operating system calls and persistent data; that being the "browser" process. This browser process contains several threads (refer to Figure 4), each with different responsibilities, but the only one we need to focus on at this level is the I/O thread. The I/O thread maintains links to all spawned processes and handles all communications with other modules (via the IPC).

Almost every time a new tab is opened in the browser, a new "renderer" process is created. Occasionally, there are a few cases when a new process is not opened for a tab, such as when a URL is opened in a fashion that would allow some script to manipulate it. Another case when this does not happen is if the number of allowed processes has been reached, in which case it starts consolidating tabs into the same process.

This renderer process is what is sandboxed for security, and it contains its own rendering engine, JavaScript interpreter, XML parser and some components from the display backend (namely Skia and GDI).

![Figure 4 – The threading model used by Chrome](image)

**Project Development**

Many of the subsystems of Chrome are designed to work without any strict dependencies. Provided that two interacting subsystems have an agreed upon interface and it provides the services an upper layer requires, the implementation details of the subsystem are irrelevant.
This is a major advantage of the layering architecture style, and allows independent teams to work on particular subsystems. For example, V8, Chrome's JavaScript interpreter is developed by a Google team in Denmark and is independent of the browser itself; V8 could be plugged into another browser provided it had an appropriate adapter to communicate with the layer above it.

The network stack of Google Chrome could also be handled by an independent team. Like V8, it only provides services to a single component above it in the hierarchy, and does not have any dependencies of its own. The team that worked on this subsystem had to provide some core functionality that the browser engine relied on, but any additional features or implementation details were not restricted.

The reverse situation is demonstrated in the Chromium project's use of the libXML library, an open-source XML parser that was not developed in house. Instead of reinventing the wheel for functionality that the Chromium team had no desire to modify, they levered their layered architecture and plugged in an existing, full featured XML parser in the same way they could have their own.

**Use Cases**

**Fetching a Basic Web Page (XHTML & CSS) Which is Not in Cache**

![Sequence diagram fetching a basic web page (XHTML & CSS) which is not in cache](image)
A URL request starts with the user entering a URL in the user interface. The URL request is sent to the browser, which then checks if the page has been stored in the browser’s cache. Assuming the page is not stored in cache, the browser connects to the network and obtains the URL data. This information is then cached for possible use in the future. The URL data is forwarded to the WebKit rendering engine, which sends to-be-rendered text and graphics to GDI and Skia, respectively. This information is rendered on an off screen "canvas". Rendered canvas information is then sent back to WebKit which then returns the bitmap to the browser where it is displayed to the user.

**Stretching a Browser Window**

![Sequence diagram for resizing the browser window to a larger viewing size](image)

*Figure 6 - Sequence diagram for resizing the browser window to a larger viewing size*

Resizing begins with the user grabbing the corner of the browser window with their mouse and dragging to extend it. This resize event is sent to the browser which checks its backing store for the most recent bitmap generated by WebKit. As the event was an expansion of the window, there will be less pixels in the backing store than needed to display the extended page view. The browser will then send a request to WebKit to generate the whole area of the page currently in view. WebKit forwards the request onto GDI and Skia which render the graphics and text, respectively. The painted canvas is sent back to WebKit which then returns the bitmap to the browser and displays the full page to the user.
Fetching a Web Page (XHTML & CSS) with JavaScript and a JavaScript Cookie

*Figure 7* - Sequence diagram for fetching a web page with JavaScript that contains a JavaScript Cookie

Fetching a webpage with JavaScript that contains a JavaScript cookie has sequences similar to the first sequence diagram. A URL request starts with the user entering a URL in the user interface. The URL request is sent to the browser, which then checks if the page has been stored in the browser’s cache. In this case, the page has been stored in cache and a network request is not made. The page data is sent to WebKit, and GDI and Skia are used to render text and graphics, respectively, as described above. JavaScript is sent to V8 to be interpreted; meanwhile, a JavaScript cookie request is made, which is forwarded to the browser. The browser handles the cookie request and returns the cookie, typically containing user preference data, to WebKit, which forwards the cookie back to V8. The remaining steps are identical to those discussed in the first use case and illustrated in Figure 5.
Conclusions

Overall, we have found that at its highest level, Google Chrome's architecture best resembles a semi-strict layering. The subsystems mainly use an object-oriented design, which is made partially evident via the team's use of C++. Overall there is a high amount of code reuse with a strong effort made to make components as modular and interchangeable as possible.

One of our biggest realizations is that Google Chrome is a substantial reinvention of the web-browser. It strays far from the typical monolithic style browser and instead makes a large effort to separate the rendering engine from the browser. Due to this layer of abstraction, the security of the browser has shown to mitigate many of the malicious coding attacks that would have affected a traditional monolithic browser.

Limitations and Lessons Learned

Throughout the duration of this assignment, our group encountered the following limitations that brought upon various challenges.

- There was a considerable scarcity of detailed architecture documentation, which pushed us to rely heavily on one source of information for the interactions: the Chromium Developers Documentation.
- The Chromium Developers Documentation described the majority of the interactions at the object level, which made it challenging to deduce the high level connections.
- In the beginning, we encountered information overload from different resources which made it difficult to distinguish what is relevant and actually used.

However, we also learned valuable lessons through this assignment that will definitely be helpful for future assignments and projects.

- Researching can be boundless. Hence, it is important to know when we have sufficient information and collective knowledge to start our analysis.
- Clear documentation is vital to understanding a system, the importance of which was made especially clear after reading through the Chromium Documentation, which, at times was not easy to comprehend.
- We were able to take advantage of the group setting as a forum to critically analyze and improve upon deductions from independent research.
Glossary

Chrome Components & Terminology

**GDI** - Graphics Display Interface: a Windows graphics renderer, used by Chrome for text rendering

**IPC** - Inter-Process Communication: the communications channel created to deliver messages between processes spawned by Chrome

**LibXML** - the open source package used by Chrome to parse XML

**libXSLT** - the open source package used by Chrome to handle XSLT

**Sandbox** - a location in memory with restricted permissions in which processes can be securely run

**Skia** - the open source package used by Chrome to render everything besides text

**V8** - an open source, lightweight and extremely fast JavaScript interpreter developed by Google for use in Chrome

**views** - Chrome's widget toolkit for creating custom browser interfaces

**WinHTTP** - the client for server-based applications in Windows Vista

**WinInet** - Windows Internet Explorer: in the supplied context, the networking protocol of it

**WTL** - Windows Template Library: an API for creating Windows native graphical user interfaces

Web Terminology

**CSS** - Cascading Style Sheets: a means of formatting the style of elements on web pages

**DOM** - Document Object Model: universal specification for laying out and providing access to (X)HTML objects

**HTML** - Hyper Text Markup Language: an encoding method used to format a web-page layout

**HTTP** - Hyper Text Transfer Protocol: the most common means for transferring web page information across the World Wide Web

**XHTML** - eXtensible Hyper Text Markup Language: a reformulation of HTML into a subset of XML

**XML** - eXtensible Markup Language: a standard for creating other markup languages and sharing data on the web, developed and recommended by the World Wide Web Consortium (W3C)

**XSL** - eXtensible Stylesheet Language: a set of technologies (XSLT, XPath & XSL-FO) used to style and manipulate XML documents

**XSLT** - eXtensible Stylesheet Language Transformations: one of the XSL technologies which enables you to manipulate data in an XML file into another format (XML or a subset of it, like XHTML)
References


