Conceptual Architecture of Mozilla Firefox 6

Presented by Fully Optimized eXperience

Group Members:
James Brereton – 06069736, 8jb66@queensu.ca
Gordon Krull – 06003108, 8gek@queensu.ca
Rob Staalduinen – 06009513, 8rjs1@queensu.ca
Katie Tanner – 06060472, k.tanner@queensu.ca

Prepared for CISC 322,
Queen's University at Kingston, Ontario, Canada

October 24th, 2011

<u>**Table of Contents**</u>

Abstract

This report intends to demonstrate our findings for the Conceptual Architecture of Mozilla Firefox 6.0. This task was completed with the intentions of being improved upon and adapted upon successfully determining the concrete architecture for this system. Throughout our research, we have discovered that Firefox takes on a layered architecture with semi-strict layering, as well as object oriented functionality. However, we have also discovered that specific sub-components, such as Necko, take on unique architectures of their own, such as a pipe-and–filter.

Throughout this report, we will examine the architecture of Firefox 6.0 as a whole, as well as provide an individual analysis of each of the major sub-systems, their specific architectures, and how these systems interact. We will also detail our derivation process, provide some examples of how this architecture would function, and explain the lessons we gathered from this process.

Introduction

Purpose of Report

In this report we discuss our findings on the conceptual architecture of Firefox 6.0. This includes our research process and our methods for deriving our final conceptual architecture and the dependencies between subsystems and components, as well as some examples data flow during common operations.

What is Firefox?

Firefox is a free, open source web browser developed and managed by the Mozilla Corporation. It currently accounts for approximately 25% of worldwide usage share of web browsers as of September 2011. It includes many advanced web browsing features such as tabbed browsing, spell checking, incremental find, live bookmarking, a download manager, private browsing, and support for adding extensions created by third party developers. It runs on various operating systems, including the big three (Windows, Mac OS X, and Linux), and is capable of running on multiple hardware platforms.

A Brief History of Firefox

Mozilla Firefox was created by Dave Hyatt and Blake Ross as an experimental branch of the Mozilla project. Firefox 1.0 was released on November 9, 2004 with further versions being released subsequently, generally on a 1-year release cycle. Version 6.0 was released on August 16th, 2011. Most of the major architecture components have remained the same throughout the development since Firefox 2.0, with changes and upgrades being implemented to subsystems and components as development moves forward.

Research Overview/Derivation Process

We began our research on the conceptual architecture of Firefox by studying the reference architecture for web browsers provided in the paper "A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser" by Alan Grosskurth and Michael W. Godfrey.

In addition to the reference architecture for web browsers, we studied the conceptual architecture of Firefox that the authors provided. Although this depicted a much earlier version of Firefox it was still instrumental in forming a basis for our understanding of the subsystems of Firefox and how they interact with one another. Using the information that we were able to extract from the paper as well as the architecture diagrams provided, we had a good starting point for our research into Firefox 6.0.

From this point, we set out to do general research on Firefox itself. This gave us a lot of valuable information on Firefox's development and functionality, the most interesting of which being that Firefox has recently transitioned to a rapid-release schedule. Once we had a general idea of what subsystems and dependencies to expect in the conceptual architecture, we then begun a more in-depth analysis of the different components of the Firefox architecture using the information we were able to gather from both the paper and our general research as a baseline.

It was at this stage, upon close analysis and research of the different components, that we were able to begin mapping out the architecture. Our main sources of documentation and descriptions of the various subsystems of Firefox were the Mozilla Developers Network (MDN) as well as the Mozilla Wiki. With this newfound knowledge, it began to become much clearer what dependencies existed within Firefox, and thus we were able to complete our conceptual architecture of Firefox.

Conceptual Architecture of Firefox

As we began the process of forming our conceptual architecture, we referred to the reference architecture for web browsers as presented in the research paper "A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser" by Alan Grosskurth and Michael W. Godfrey. We began to conceptualize our idea for the architecture of Firefox 6 through the reading of all available documentation on the different components and subsystems, provided by Mozilla as well as other developers. Our findings on the architecture of Firefox consisted of a Virtual Machine Style Layered Architecture, with XPCOM serving as the crucial component that allows the system to function as a whole. Our conceptual architecture for Firefox 6 is depicted on the following page.

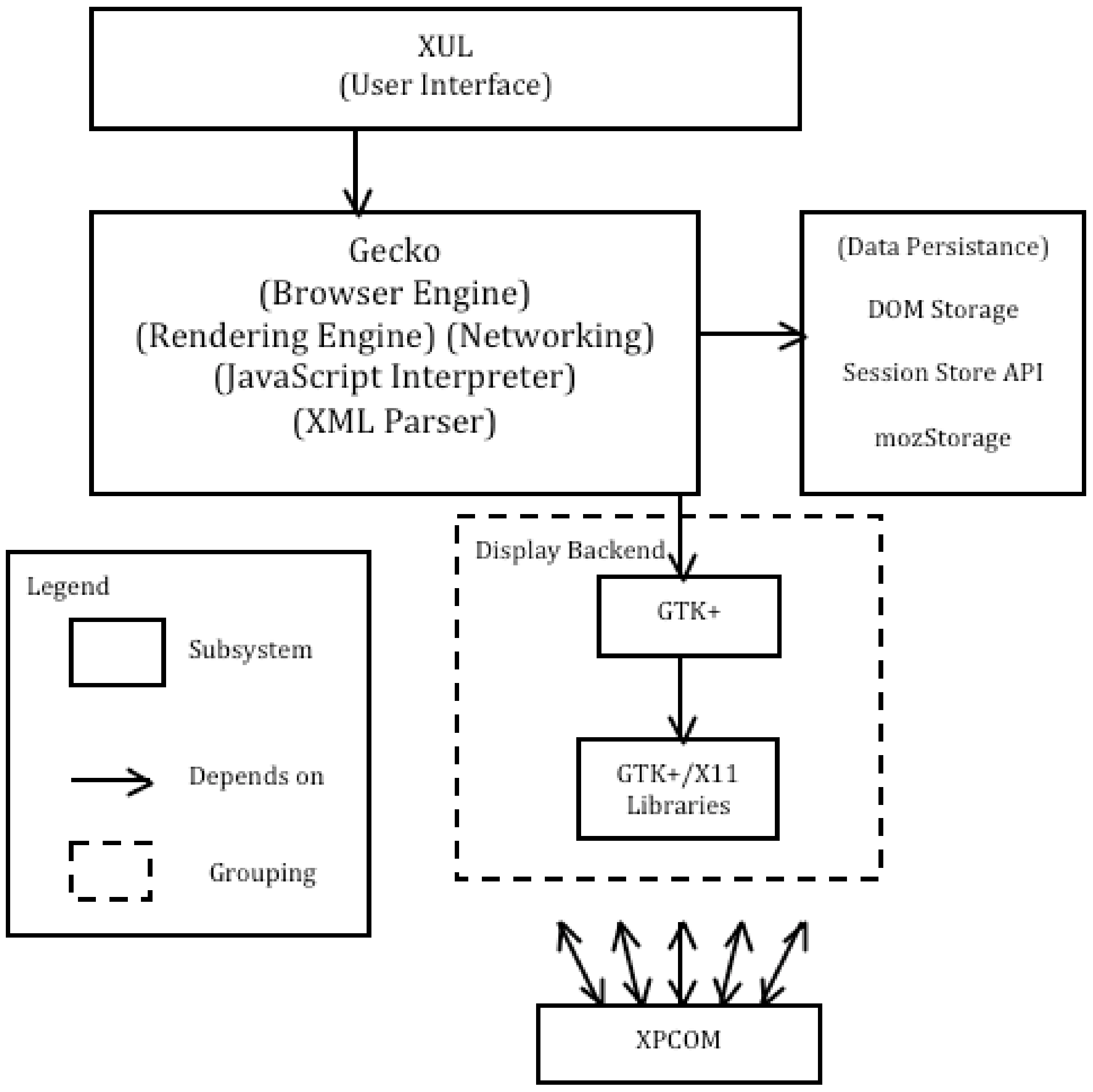

Figure 1: Conceptual Architecture of Mozilla Firefox 6.0

While reading the documentation, we discovered that although in previous versions of Firefox key subsystems such as Necko and SpiderMonkey were considered separate from Gecko, they are now considered components of the Gecko Subsystem. While Gecko's primary purpose remains to render and display WebPages to the user through the User Interface, it now is considered to contain these subsystems and therefore by using the reference architecture we define Gecko as not only the Browser Engine and Rendering engine, but the Networking, JavaScript Interpreter, and the XML Parser components as well.

The Layered Architecture of Firefox clearly indicates the dependencies and how FireFox functions. The topmost layer consists of the user interface, which depends upon the rest of the system to render the browser. This layer serves as the main venue through which the user interacts with the Firefox system. The UI calls upon the second layer, which service the UI layer with rendering and data access. This layer contains both the Gecko and Necko subsystems, which are also implemented as a Virtual Machine layered architectural style with pipe and filter elements. Finally, the lowest layer is the Display Backend. The Display Backend provides proper GUI formatting information to Gecko so that pages can be properly rendered on any platform according to that platform's specifications.

The nature of a layered architecture benefits the system by allowing it great portability since the lowest layer provides platform-specific rendering for the system. The system is also highly modifiable since there are many different components and each performs a specific task.

Having presented our derived conceptual architecture for Firefox, we will now describe each component of the Firefox system as well as how the various components interact.

XUL and XULRunner

As previously mentioned, the topmost layer of Firefox is its user interface, which corresponds to the top layer of the reference architecture. XUL, XML UI Language, provides the basis for Firefox's user interface. Rather than being hardwired into the application itself, the UI is loaded from a separate UI description written in XUL. XUL is essentially XML (extensible markup language), but allows for HTML elements, including JavaScript, to be incorporated and has a specific definition for a few element types. Mozilla uses this to build cross-platform applications such as web browsers and mail clients. Proper UI descriptions must take into account various platforms' differing layouts and elements such as dialog boxes. While a single cross-platform UI description could function on different platforms, build engineers and UI designers must maintain platform-specific versions of some XUL documents so that the browser can be rendered in the optimal format for each platform.

XULRunner is the runtime environment used for the deployment of XUL applications such as Firefox, or an application written in any language supported by the Mozilla web platform (such as HTML, XHTML, SVG, or XUL). Among other things, it includes the API and User Interface for installing, upgrading, and uninstalling Firefox as well as lixbul, a solution for embedding Mozilla technologies in other projects.

The user interface layer relies on Gecko for its rendering and parsing capabilities as well as for Necko, the networking component. The UI layer is also dependent on XPCOM.

Gecko

As mentioned above, the Gecko subsystem now serves as more than just Firefox's Browser Engine and the Rendering Engine. Its components now include the JavaScript Interpreter, SpiderMonkey, and the Networking Subsystem (Necko). Gecko truly serves as the heart of the Firefox browser. It is highly standard compliant using many different standards (HTML, DOM, RDF, XML to name a few) and is used in many different Mozilla products due to its capabilities and strengths when it comes to fast and effective rendering. Below is our conceptual architecture for the Gecko subsystem, for which we have derived an object-oriented architecture, displaying the dependencies between each of its components. This should provide an indication as to how the various components of Gecko interact and function, but we will describe each component in detail following the conceptual architecture diagram.

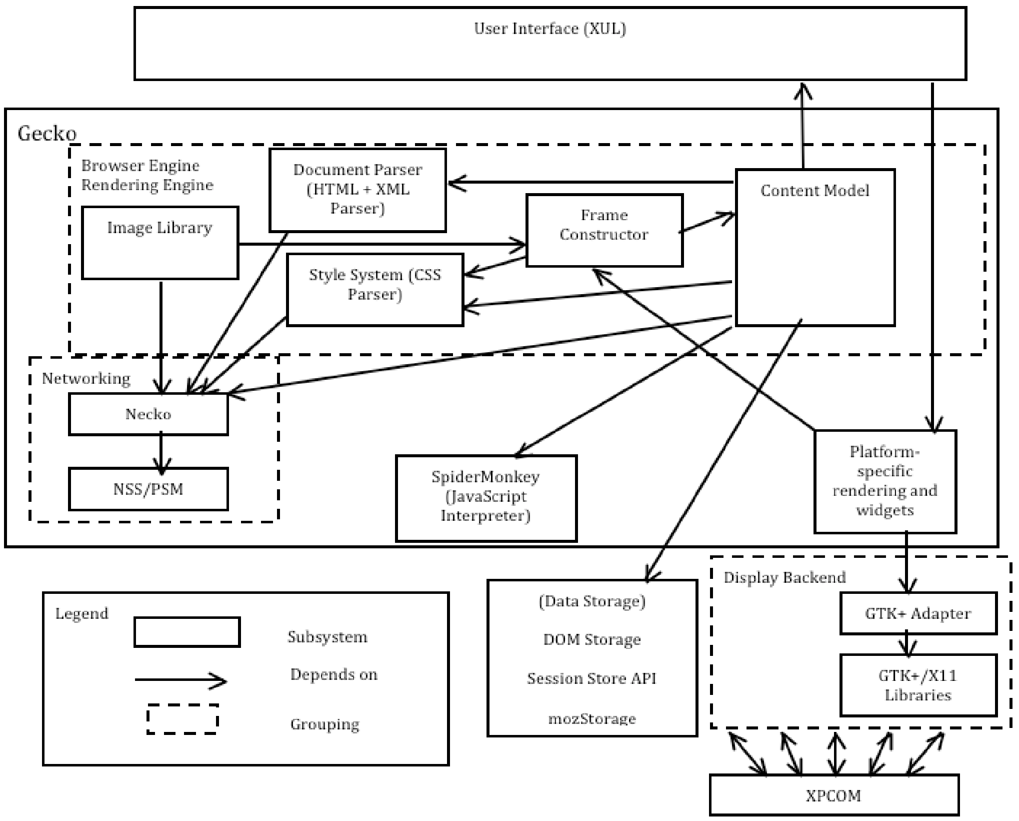

Figure 2: Conceptual Architecture of the Gecko subsystem

We have divided up each of the components into the groups to clearly distinguish the different components and their functions within the gecko subsystem. We have partitioned components into one group for the Browser and Rendering Engine aspect of Gecko, and another for Networking. Not included in these groupings but present in Gecko are the JavaScript interpreter and a component for platform-specific rendering.

Browser Engine/Rendering Engine Components

Document Parser (HTML & XML Parser)

In Firefox 6, the Document Parser component serves as the parser for both HTML and XML. The Document Parser accesses the HTML code via Necko after receiving the URL data from the User Interface through the Content Model. After parsing the HTML code, it sends the code to the Content Model for further manipulation before passing it to the Frame Constructor for rendering of the web page. The Document Parser also contains the Expat Library for XML, which allows it for the parsing of any XML data received from Necko before the code is sent to the Content Model.

Content Model

The Content Model receives the aspects of each fetched web page from the Document Parser for manipulation before sending the manipulated code to the Frame Constructor to actually construct the webpage. It begins by interacting with the user interface and receiving URL data before sending this information to Necko to retrieve the HTML code as well other components required for the requested webpage. It then manipulates the data for rendering through the use of a DOM (Document Object Model) tree. It also interacts with SpiderMonkey for any JavaScript interpretation that is required, the Image Library for any image data that the webpage requires, and the components of the Data Persistence subsystem for any cached data that can be used in the formation of the web page. The Content Model also relies on Data Persistence when retrieving user data such as preferences and bookmarks. Once the content model has received all the necessary data, it passes the manipulated data from the DOM tree to the Frame Constructor.

Style System (CSS Parser)

The main purpose of the Style System within Gecko is to parse any CSS data received from Necko. The data is then sent to the Frame Constructor to properly render the data through the User Interface. If more information is required for the CSS data, it is received from the Content Model.

Image Library

The Image Library retrieves a webpage's image data from Necko and loads it before sending it to the Content Model.

Frame Constructor

The Frame Constructor is the subsystem that receives all of the data necessary to construct a web page before it is sent back to the user interface layer for the user to view. This includes parsed HTML, XML, and JavaScript, from the Content Model and image data from the Image Library. It begins by receiving the DOM Tree data from the Content Model and any CSS data from the Style System. The Frame Constructor then proceeds to build the Web Page, using the Platform-Specific Rendering and Widgets component to construct the webpage in the manner best suited for the platform for which it is being requested. The Frame Constructor sends its constructed page to the subsystem for Platform-Specific Rendering before the page is finally displayed to the user.

SpiderMonkey

SpiderMonkey, is FireFox's JavaScript engine, written in C/C++. Essentially, it is a fast interpreter that operates on a full range of JavaScript values. Its components are an interpreter, a compiler, 2 JIT or just-in-time compilers, a decompiler, garbage collection, and a standard library. SpiderMonkey receives JavaScript code to be interpreted before sending the interpreted code back to the Content Model for further rendering. SpiderMonkey also contains a few public APIs so that other applications can utilize SpiderMonkey for JavaScript support.

Platform-Specific Rendering and Widgets

Gecko includes this subsystem, which provides platform-specific GUI data to the Frame Constructor. It interacts with the Display Backend, specifically the GTK+ adapter. It also serves as the final step in rendering a page before it is displayed to the user.

Networking

Necko

Necko is the main networking library of Mozilla Firefox, responsible for the transport of data from a location on the internet to the other components of Firefox, which will render the data into a form that can be displayed by the User Interface. Necko is able to transport data through the commonly used locator known as URL(Uniform Resource Locator). Necko has protocols to handle many different types of data, including http and https, ftp, and files to name a few.

Architecture: Necko itself has a rather unique architecture within the Gecko subsystem. It uses a pipe-and-filter style architecture for its control flow, which allows for the logical handling of the different protocols available. The dependencies of the Necko Architecture can be seen below.

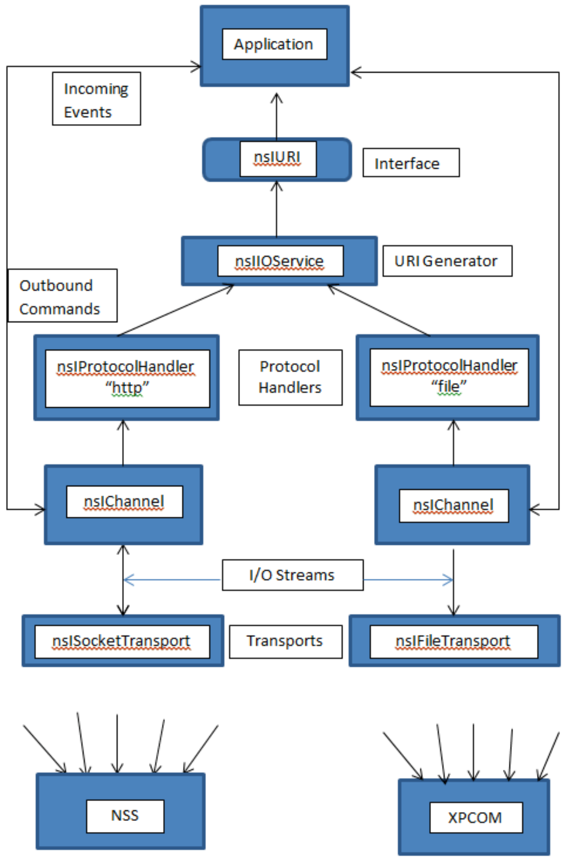

Figure 3: Conceptual Architecture of Necko

nsISocketTransport & nsIFileTransport are the two main transport interfaces of Necko, allowing for the physical transport of data. These transports pass the data to an IO Stream

nsIChannel provides an interface for the movement of data to and from a URL. There is a direct 1-to-1 ratio between nsIChannels and URLs being accessed. It depends on the nsIProtocolHandler to create a channel with the proper protocol. The nsIChannel provides the actual data flow to the rest of Firefox

nsIProtocolHandler allows for the creation of channels with the proper protocols. For example the http protocol handler creates http nsIChannels

nsIIOService acts as a protocol lookup, as a URL has no inherent knowledge of what protocol it represents. It provides the proper protocol to the protocol handler in order to interact with the specified URL.

Necko relies on the XPCOM libraries for interacting with other system components, and the NSS libraries for its network security.

Network Security Services (NSS)

Network Security Services consists of a set of libraries, APIs, utilities and documentation designed to support cross-platform development of security-enabled client and server applications. It supports a wide variety of security standards, such as SSL v2 and v3, TLS, PKCS #5, PKCS #7, PKCS #11, PKCS #12, S/MIME, X.509 v3 certificates, and others. The Necko subsystem depends on NSS in order to handle any secure data that it encounters while loading a webpage.

Personal Security Manager(PSM)

The Personal Security Manager is built on top of NSS and consists of a set of libraries that perform cryptographic operations on behalf of a client application. These can include setting up an SSL connection, object signing and signature verification, certificate management (issuing and revoking) and other common functions.

Data Persistence

The purpose of the data persistence subsystem is to store user, tab and window data across Firefox sessions. This can be achieved in different ways using the different components contained within the data persistence subsystem. In addition to this XUL provides a mechanism for helping to store persistent data for remembering the state of windows, toolbars, and more. An attribute called "persist" is used to determine which attributes to save. The information is then collected and stored in a file in the same directory as other user preferences. XUL allows for saving the state of any element.

In our conceptual architecture you can see that the data persistence is in the 2nd layer, the same as the Gecko subsystem. The data persistence has no outgoing dependencies to other subsystems in the architecture and only has one incoming dependency from Gecko. More specifically we have found that the dependency comes from the Content Model component within Gecko, which depends on the data persistence when retrieving bookmarks, user preferences, and cached website data.

Session Store API

The session store API makes it possible for extensions to easily save and restore data across Firefox sessions using a simple API that allows access to the session store feature. One instance where supporting this feature is a major benefit for an extension is that from Firefox 2.0 and onwards have allowed the user to undo the closing of tabs. In order to restore the tab correctly an extension must use methods provided by the session store API in order to save any data that it will need to be restored, and to retrieve the previous data when the tab is restored. Two important interfaces for the session store API are nsISessionStore and nsISessionStartup. The session store API is implemented using nsISessionStore, which provides a means for extensions and other code to store data in associations with browser sessions, tabs and windows and nsISessionStartup actually handles the session restore process.

DOM Storage

DOM Storage is designed to provide a larger, more secure, and easier-to-use alternative to storing information in cookies and was first introduced in Firefox 2. It is a means through which string key/value pairs can be securely stored and later retrieved for use. The DOM Storage is very useful because there are no good browser-only methods that exist for persisting reasonable amounts of data for any period of time. For example browser cookies can only hold a limited amount of data and provide no organization for persisted data, and other storage methods require external plugins. One of the major benefits of using the DOM Storage is that it supports more advanced abilities. For example, it a user to "work offline" for extended periods of time

Storage(mozStorage)

Storage (previously called mozStorage) is a SQLite database API that is available to trusted callers. This means Firefox extensions and Firefox components only. It handles the process for getting the storage service and then connects to databases and executes statements on the open database connection. It is able to execute a statement either synchronously or asynchronously. Beginning with version 3.0, Firefox uses Storage for its history, form history and bookmark data.

XPCOM

XPCOM, the Cross Platform Component Object Model, allows developers to create cross-platform, modular software. The primary purpose of XPCOM is to allow developers to modularize large software projects into smaller components and thereby allow these components to be developed and built independently of one another. The components, which are contained in reusable binary libraries, are then re-assembled at runtime of the application.

XPCOM also provides the tools and libraries that enable the loading and manipulation of these modularized components. Additionally, it provides much of the same functionality as a development platform, including component management, file abstraction, object message passing, and memory management. Although XPCOM provides its own core components and classes, the majority of XPCOM components are provided by other parts of Firefox such as Gecko and Necko, hence its outgoing dependencies.

XPCOM is used by every component of Firefox as it is the basis for manipulating the various components and objects within the system. However, its most important use is within Gecko, because XPCOM provides the means of accessing the Gecko library functionality as well as the means of embedding or extending Gecko. XPCOM's modularizing of Firefox greatly enhances the system's performance, modifiability, maintainability, as well as ease of development.

Display Backend - GTK+ Adapter and the GTK+/X11 Libraries

The lowest layer of Firefox's Architecture contains both the GTK+ Adapter and the GTK+/X11 Libraries. Both GTK+ and X11 are multiplatform GUI widgets and toolkits used to generate GUIs. The GTK+ Adapter and the GTK+/X11 Libraries form what is considered in the Firefox architecture the Display Backend. This layer provides Gecko's Rendering Engine with platform-specific graphical data to ensure proper rendering on each individual platform and is responsible for the high level of portability of the Firefox browser.

Sequence Diagrams

Below are two sequence diagrams demonstrating data flow between systems in Firefox for two very common functions of a web browser. We have included one sequence diagram which depicts data flow when displaying a cached webpage, and another for displaying a non-cached webpage.

Displaying a cached webpage

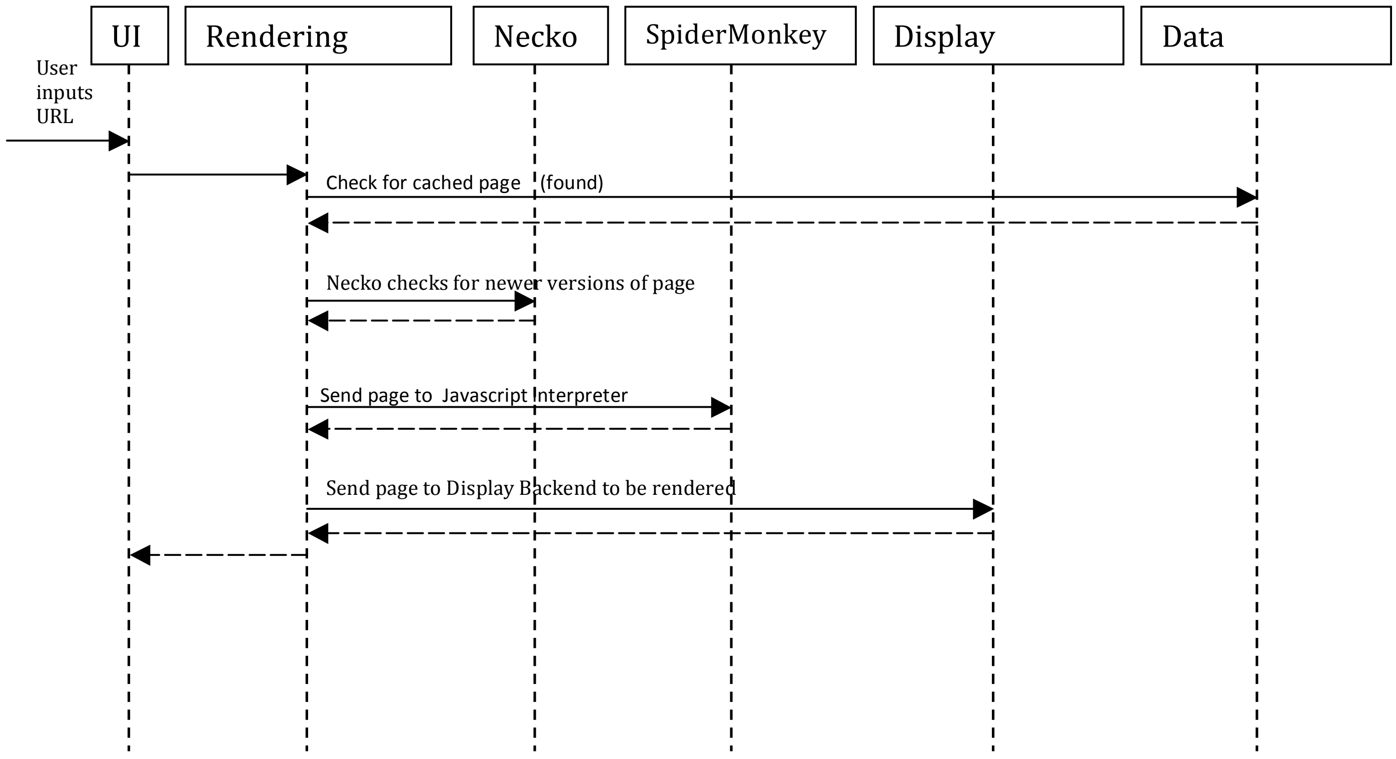
UI
Rendering
Necko
SpiderMonkey
Display
Data
User inputs URL
Check for cached page (found)
Necko checks for newer versions of page
Send page to Javascript Interpreter
Send page to Display Backend to be rendered

Figure 4: Sequence Diagram for Displaying a Cached Webpage

This sequence diagram illustrates the process of displaying a cached webpage. The process begins with the user inputting a URL to the User Interface (UI) component of Firefox. The UI then sends this to the Rendering Engine within Gecko. The Rendering Engine checks with the Data Persistence component to see if there is a cached version of the requested page stored and, upon finding that the page exists within the cache, sends the page back to the Rendering Engine. The networking subsystem, Necko, then checks the web to see if there is a newer version of the page available and reacts accordingly (i.e., fetches the newer page version if required) and then sends the results of its findings back to the Rendering Engine for further processing. The Rendering Engine now begins the process of actually rendering the page by sending it to SpiderMonkey to interpret any Javascript that needs to be displayed. SpiderMonkey returns the interpreted page to the Rendering Engine, which in turn sends the page to the Display Backend to finalize the rendering. Finally, the page is sent back to the Rendering Engine, then to the UI to be displayed for the user.

<u>Sequence Diagram – Displaying a non-cached webpage</u>

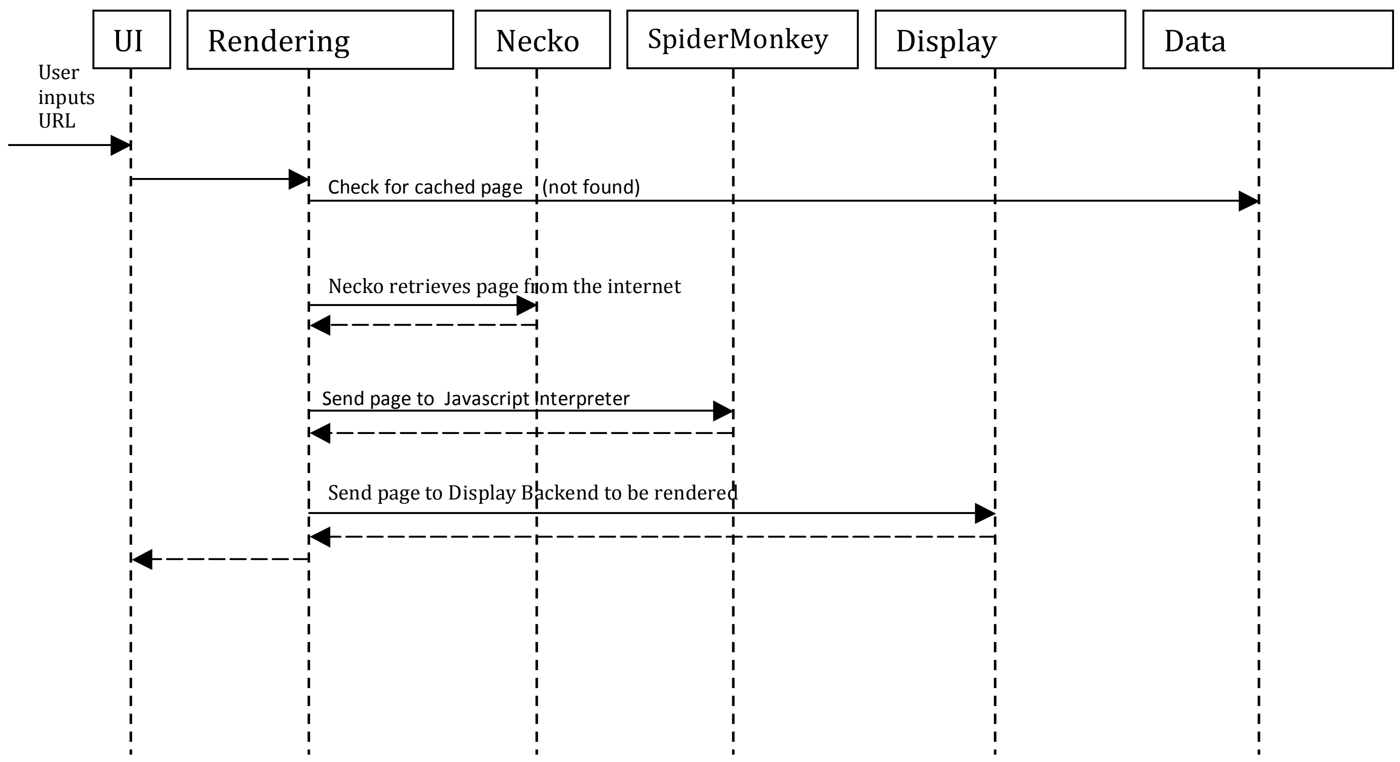
UI
Rendering
Necko
SpiderMonkey
Display
Data
User inputs URL
Check for cached page (not found)
Necko retrieves page from the internet
Send page to Javascript Interpreter
Send page to Display Backend to be rendered

Figure 5: Sequence Diagram for Displaying a Non-cached Webpage

This sequence diagram depicts data flow when displaying a non-cached webpage. The process begins with the user inputting a URL through the User Interface (UI) component of Firefox. The UI then relays this to the Rendering Engine within Gecko. The Rendering Engine checks with the Data Persistence component for a cached version of the requested page and finds that one does not exist. The networking subsystem, Necko, then fetches the requested page from the web and sends the page back to the Rendering Engine once it is found. The Rendering Engine now begins the process of actually rendering the page by sending it to SpiderMonkey to interpret any Javascript that needs to be displayed. SpiderMonkey returns the interpreted page to the Rendering Engine, which in turn sends the page to the Display Backend to finalize the rendering. Finally, the page is sent back to the Rendering Engine, then to the UI to be displayed for the user.

Lessons Learned

During the course of this project, we learned many things about the process of deriving a conceptual architecture. First and foremost, we learned that the documentation base for an open source project such as Firefox can often be limited, as it can easily be deemed as a low priority. To make matters worse, the new rapid release cycle Firefox is adopting means that the overall operation of Firefox itself is changing rapidly, making the documentation often inconsistent.

We also learned the importance of understanding how each component links together. We initially split up each component between our group members, in order to examine each in depth. Though this gave us a better opportunity to master the specific components, it was when we came together and discovered how they worked as a whole that we gained a much deeper understanding of the individual components and their interfaces between each other.

Conclusion

The overall architecture of Firefox 6.0 was a layered architecture with semi-strict layering, with an object oriented style of control for the Gecko subsystem. One of the interesting changes of the current build of Firefox is that many components, such as the networking interface(Necko and NSS), the Javascript Interpreter(Spidermonkey) and the XML Parser(Expat) have been integrated in to Gecko, making in truly the heart and soul of the Firefox system.

Firefox remains an outstanding example of open-source software, with a great focus on cross platform usage, as evidenced by the XPCOM library, which allows for ease of use on all platforms. As well, the fact that each of its components (XUL, Gecko, Necko, SpiderMonkey, Expat, Data Persistence, XPCOM and GTK+) remaining as distinct objects allows for an easily upgradable and modifiable system.

The next objective of this assignment will be an in depth examination of the source code, allowing for us to refine our conceptual architecture and derive the concrete architecture of Firefox 6.0.

References

- https://developer.mozilla.org/en/Mozilla_Application_Framework_in_Detail
- https://developer.mozilla.org/en/Necko
- https://developer.mozilla.org/en/Necko_Architecture
- http://www.w3counter.com/globalstats.php
- https://developer.mozilla.org/en/Creating_XPCOM_Components/An_Overview_of_XPCOM
- https://developer.mozilla.org/En/SpiderMonkey/Internal
- https://developer.mozilla.org/en/DOM/Storage
- https://developer.mozilla.org/en/Session_store_API
- https://developer.mozilla.org/en/XPCOM_Interface_Reference/nsISessionStore
- https://developer.mozilla.org/en/XPCOM_Interface_Reference/nsISessionStartup
- https://developer.mozilla.org/en/XUL_Tutorial/Persistent_Data
- https://developer.mozilla.org/en/NSS
- http://www.mozilla.org/projects/security/pki/nss/
- https://developer.mozilla.org/en/NSS_FAQ
- http://www.mozilla.org/projects/security/pki/psm/
- https://wiki.mozilla.org/Gecko:Home_Page
- https://wiki.mozilla.org/Gecko:Overview
- https://developer.mozilla.org/en/Gecko_FAQ
- http://research.cs.queensu.ca/home/emads/teaching/readings/emse-browserRefArch.pdf
- http://www.gtk.org/
- https://developer.mozilla.org/en/Gecko_Embedding_Basics
- http://xkcd.com/198/
- https://wiki.mozilla.org/Firefox/Roadmap
- http://www.x.org/wiki/
- http://en.wikipedia.org/wiki/Firefox