Conceptual Architecture of Firefox

October 26, 2015

Drew Noel	drewmnoel@gmail.com	212513784
Einas Madi	emadi@my.yorku.ca	211558897
Siraj Rauff	srauff@my.yorku.ca	212592192
Skyler Layne	skylerclayne@gmail.com	212166906

Table of Contents

Table of Contents 2
1. Abstract 3
2. Introduction and Overview 3
3. Architecture 5
   3.1 User Interface (UI) Widgets and Services 6
   3.2 Content (DOM) Layer 6
   3.3 Core Services Layer 8
   3.4 Data Flow among Parts 8
4. External Interfaces 9
5. Use Cases 9
6. Glossary 11
7. Conclusions 11
8. Lessons Learned 12
   8.1 Existence of Documentation 12
   8.2 Updating Documentation 12
   8.3 Planning Future Extendibility 13
   8.4 Use of Gecko 14
9. References 15

1. Abstract

This project involves the derivation of the Mozilla Firefox conceptual architecture employing various methods mentioned throughout the course. Such methods include observation of a reference browser architecture - which Firefox was found to follow quite closely - as well as consultation of the Mozilla Developer Network Documentation. Due to lack of proper documentation during the early period of Firefox's development as well as a lack of maintenance since the initial creation of the documentation, it was necessary to consult the source code in order to determine the validity of a document.

The architecture of Mozilla Firefox is based on the Mozilla Application Framework, which is used by many Mozilla applications including Firefox, Thunderbird and Firefox OS. At the top level, the framework consists of a layered-architectural style comprised of three main layers:

- User Interface (UI) Widgets and Services
- Content (DOM)
- Core Services

Each of these layers was found to have its own unique architectural style. The top layer makes use of Implicit Invocation for event handling, as well as Object-Model and adapter architectures together for various components including extended functionality such as extensions. The second layer was found to be a pipeline-repository hybrid for rendering of web-page content. The final layer also makes use of Object-Oriented style for the numerous services it uses including networking and native graphics.

Some notable discoveries throughout the derivation of the architecture include both the lack of proper documentation techniques throughout the initial period of Firefox's development, as well as a stark contrast relative to the newer products of the Mozilla Foundation such as Firefox OS. Much of the documentation was broken, out-of-date or missing entirely. This seems to have improved, as newer documentation is both informative and up-to-date; however, the Firefox OS project is considerably younger so how this holds up over time remains to be seen.

Another concerning development was found in the content rendering pipeline in Firefox which remains the last browser to make use of single-threaded rendering, putting the browser at a notable disadvantage compared to competing modern browsers.

2. Introduction and Overview

Firefox is an open source project maintained by the Mozilla Foundation, a non-profit organization that heads a multitude of open-source projects. The Mozilla Foundation

breaks down all of its projects into several divisions: coding, testing, activism, localization, helping, writing, learning, teaching, fellowship, and metrics. Each of these divisions are divided into several categories. For example, the coding division is divided into General Firefox, Firefox OS, the Firefox Add-ons project and more.

Each of these categories are managed by a steward who hosts weekly meetings for his team as well as any volunteers who wish to listen or participate. In these meetings, ideas for improvement and advancement are proposed and discussed. It is the steward, ultimately, who approves or disapproves ideas. If approval is granted, certain individuals are tasked with overseeing the implementation of the proposed changes. Upon completion of the assignment, the submitted code is reviewed and tested in isolation for any bugs before merging the new code with the project's main code.

The Mozilla Firefox project makes use of *BugZilla*, a server software for users to either search for reported bugs, or report new ones. Bugs that have been reported are categorized by components and severity. They are then assigned to volunteers who have registered with the Mozilla Foundation and have indicated interest for bugs in specific categories.

In this report, we will take a top-down approach of the conceptual architecture of Firefox, talking about each of the layers of Firefox including their inner architectures and components as well as their purpose. Examination of use-cases as well as interactions between the various components comprising Firefox provide us with a general idea of the conceptual architecture of Firefox as well as any drawbacks or advantages of the design.

In analyzing Firefox, we discovered that it is a layered architecture consisting of the User Interface (UI) layer, the Content (DOM) layer, and the Core Services layer. The first layer incorporates the adapter pattern and uses the object-model architecture. The second layer is powered by the Gecko engine, which uses a hybrid pipeline-repository architecture. The third layer is comprised of objects written in native C/C++ or Java that utilizes the XPCOM, which itself makes use of an object-oriented architecture.

3. Architecture

The Mozilla Firefox architecture follows the same architecture as other Mozilla Application Framework apps with a layered top-level architecture (Tiner, Susan D. et al.). This framework very closely follows the reference architecture for web browsers, seen in figure 3.1. It is used in every Mozilla Framework Application including Firefox for both iOS/Android, Firefox OS, Thunderbird, etc.

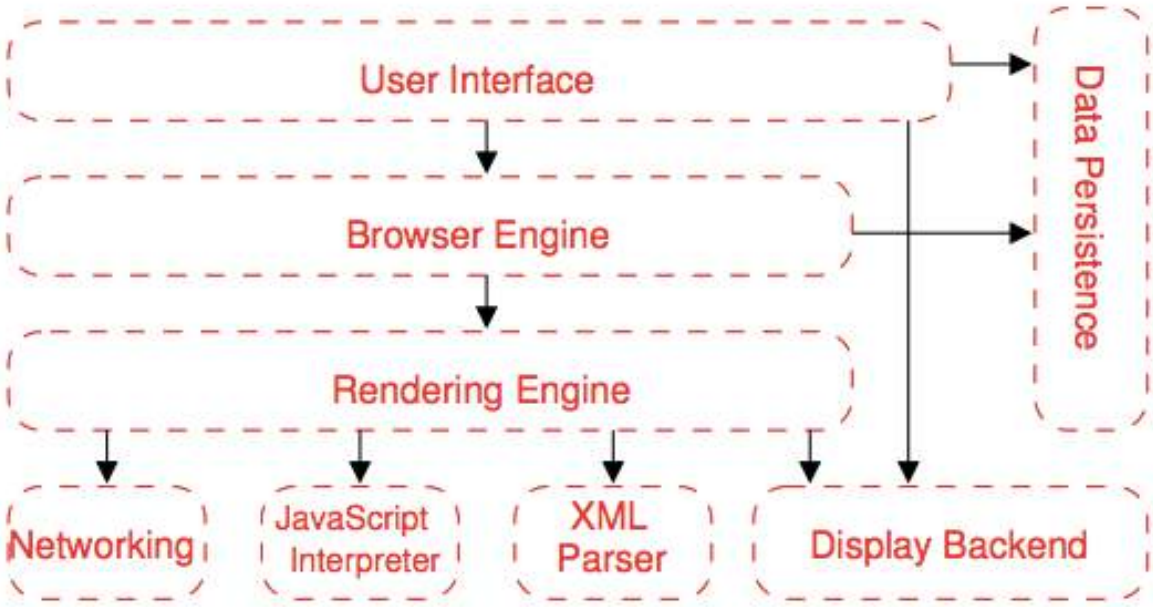


Figure 3.1 Reference Browser Architecture (Grosskurth and Godfrey)

Some key differences between Mozilla Firefox's architecture and the reference browser architecture include the unification of certain layers. This results in a three-layered approach in contrast to the four distinct layers and data-persistence repository in the reference architecture (Angus et al.).

As can be seen from the corresponding figures 3.1 and 3.2, the user interface and data persistence layer are consolidated into the XUL/UI layer in the Mozilla Application Framework.

The browser engine and rendering engine in the reference architecture are also combined; they are both handled by Gecko. The remaining services, including networking services and the JavaScript interpreter, comprise an entire layer in Firefox known as the 'Core App Services' layer which makes use of object-oriented architecture to enable pluggable modules.

It is worth noting that each of these layers in Firefox (and by extension the Mozilla Application Framework's layers) have their own unique architectural style.

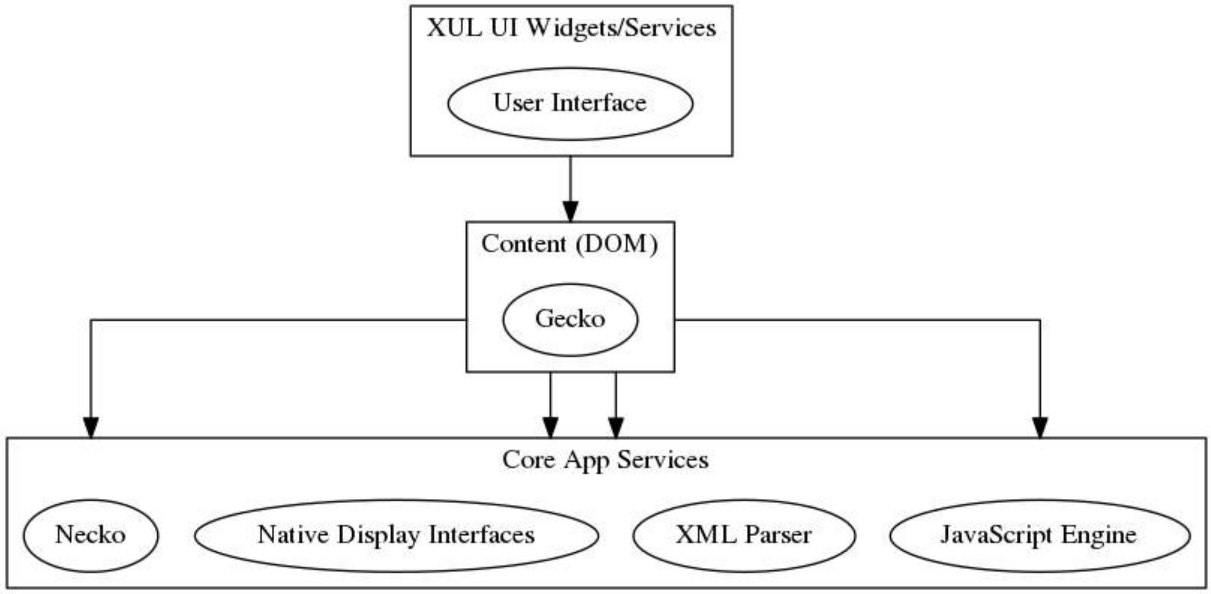


Figure 3.2 High-level Architecture of Mozilla Firefox

3.1 User Interface (UI) Widgets and Services

The top-most layer is known as the User Interface/Widgets and Services layer. This layer is created using XML User Language (XUL), a Mozilla invention that allows the interface to be designed and programmed using the same language constructs that are used to make web pages. Functionality in this layer is controlled via XML Binding Language (XBL). XBL specifies both, visual and behavioural characteristics of XUL objects. Interaction with the middle layer occurs via event triggers, specified by the Content (DOM) Layer.

This separation of content from the core engine of the browser allows easy modification of the browser's UI as well as easy extensionality through extensions/apps. In addition, this has been essential in creating a cross-platform web browser.

3.2 Content (DOM) Layer

The second layer consists entirely of a single object and the main driving force behind Firefox, the Gecko engine. The engine handles the rendering and layout of web content as well as the user interface due to its nature (namely that the UI is written in XUL). Communication between the top layer and the middle layer is accomplished via DOM

event triggers. These triggers are similar in nature to JavaScript's event triggers, due again, to the similarity between the user interface and web pages.

While Firefox itself is a multithreaded browser, some parts of the architecture do not support multithreading (Baron and Watt). An example is found in the Gecko rendering engine as displayed in figure 3.3. The reasoning behind this is because when multithreaded was attempted, it would traverse the document and parse it in different ways leading to inconsistent results (Young). An asynchronous event loop was then chosen to mitigate this and provide reliable consistent results. The loop runs in a hybrid pipeline-repository architecture in order to pass the necessary information around.

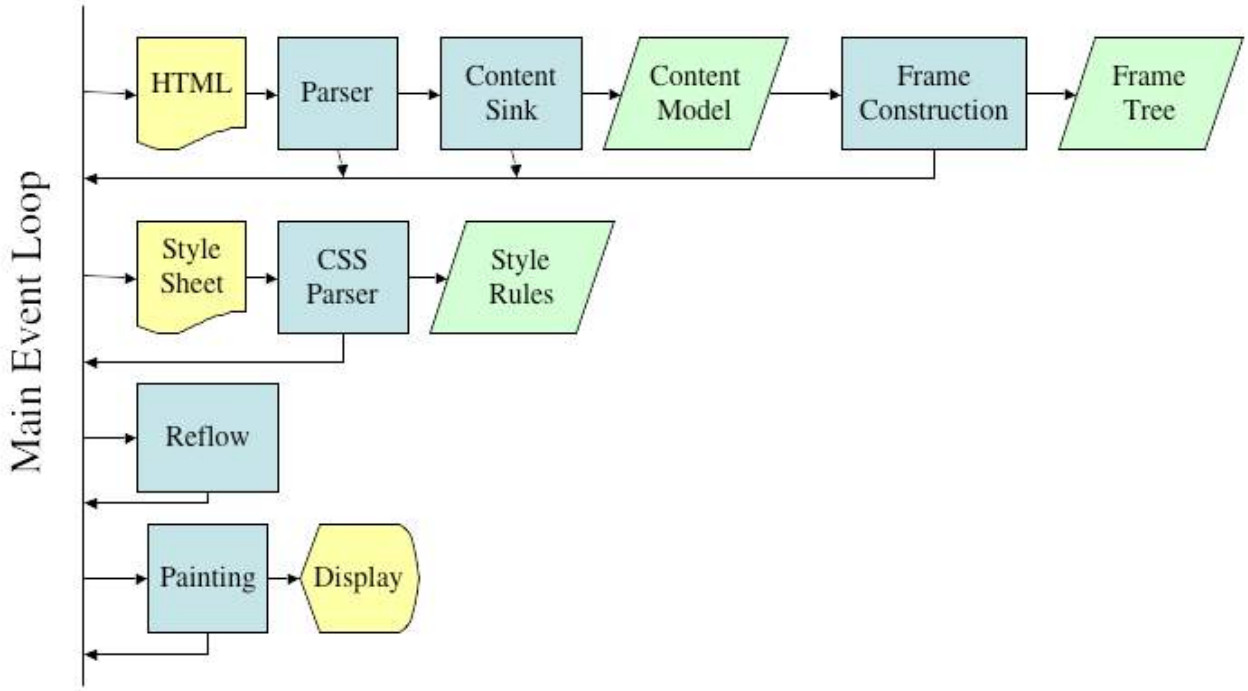


Figure 3.3 Gecko Document Parser (Waterson, Chris et al.)

Interesting to note is that the Gecko engine itself does not care whether the data came from the UI or from the network (Hobson, Stephanie et al.). The UI layout and the specific XPCOM objects chosen are the main differences between the Firefox browser and the Thunderbird email client (Fuqiao, Xue et al.).

Any data sent to Gecko first goes through its HTML Parser. Once the parsing is complete, the information is sent to the DOM or Content Model, which arranges the parsed data into a tree known as the DOM tree or content tree. The content tree is then put into frames and arranged into a Frame Tree using the Frame Constructor. A *frame* is an abstract box a DOM element is displayed within and all the related elements that the document needs to display the data (Evans, Ellen et al.). Each of these frame

elements have a pointer that leads back to the Content Model so that any changes in the Content Model result in a change to the Frame Tree.

New information is constantly being obtained using a process called reflow; this is a recursive process which allows the Frame Tree to be continuously modified in order to accommodate new data as it is received. Once the Content Model is updated and the Frame Tree is complete, the data is sent to the View Manager. The View Manager parses the data, calculates where elements need to be drawn, and sends it to the painting process, which draws the corresponding elements.

3.3 Core Services Layer

The bottom layer consists of the core app services and notably supports multi-threading. These app services are written as native C/C++ or even Java XPCOM objects and interface with the second layer where needed.

The XPCOM is an object-oriented architecture that is used in the bottom layer. The core app services may also be programmed in various scripting languages such as JavaScript using XPConnect. XPConnect makes use of the adapter pattern to allow scripts to interface with Gecko and other XPCOM objects as if they were native C/C++/Java objects. Some of the XPCOM objects that are included in Firefox are:

- Necko, the network module
- Native display interfaces that deal with painting the display
- An XML parser
- SpiderMonkey, the JavaScript engine

An important concept to note here is that each of the core app services may have its own thread.

3.4 Data Flow among Parts

All events begin in the UI layer of Firefox where a DOM event is triggered during user interactions with UI components. The event information is transferred to the Gecko Engine, which then requests the document from either Necko (the network module) or a local source depending on the request. The data is then parsed and managed by Gecko's rendering engine and when complete, returns the data to the UI to update its display.

4. External Interfaces

Firefox's communication with the outside system is restricted to the XPCOM modules as the below diagram illustrates. This intentional design choice allows the development of the user interface and the Gecko engine to be completely separate from the system communication. This aids in Firefox being compiled on any platform, since platform-specific code will all be kept in the XPCOM modules themselves. In addition, this allows the rapid implementation of new functionality to the Gecko engine.

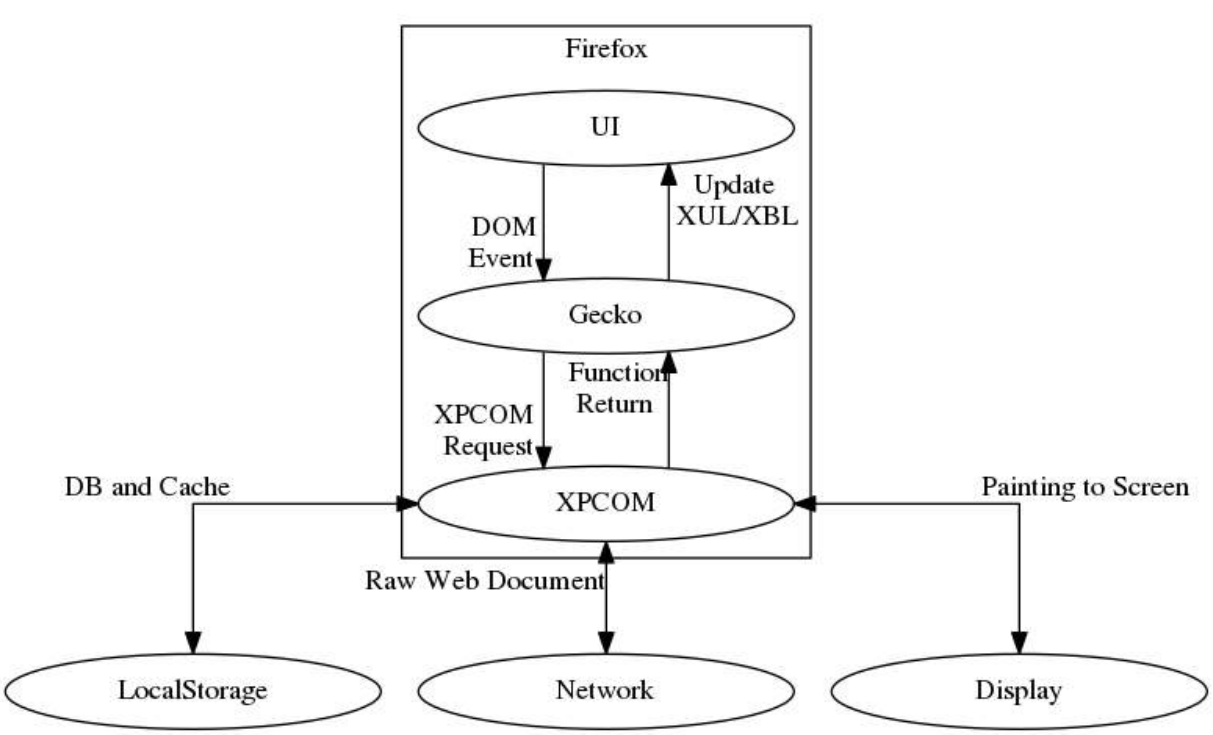


Figure 4.1 System Overview

5. Use Cases

Here we are given some use cases that show the communication between various components as well as overall order of execution.

The sequence diagram in Figure 5.1 displays the data flow for a non-cached page. The sequence starts when a user enters a URL through the UI. The UI relays this data to the rendering engine in Gecko, which then sends the requested URL to its Data Persistence component to check for a cached version of the entered URL; in this case, it returns nothing. The rendering engine then sends the requested URL to Necko, the network library, to retrieve the page from the internet assuming it exists. Necko returns the to

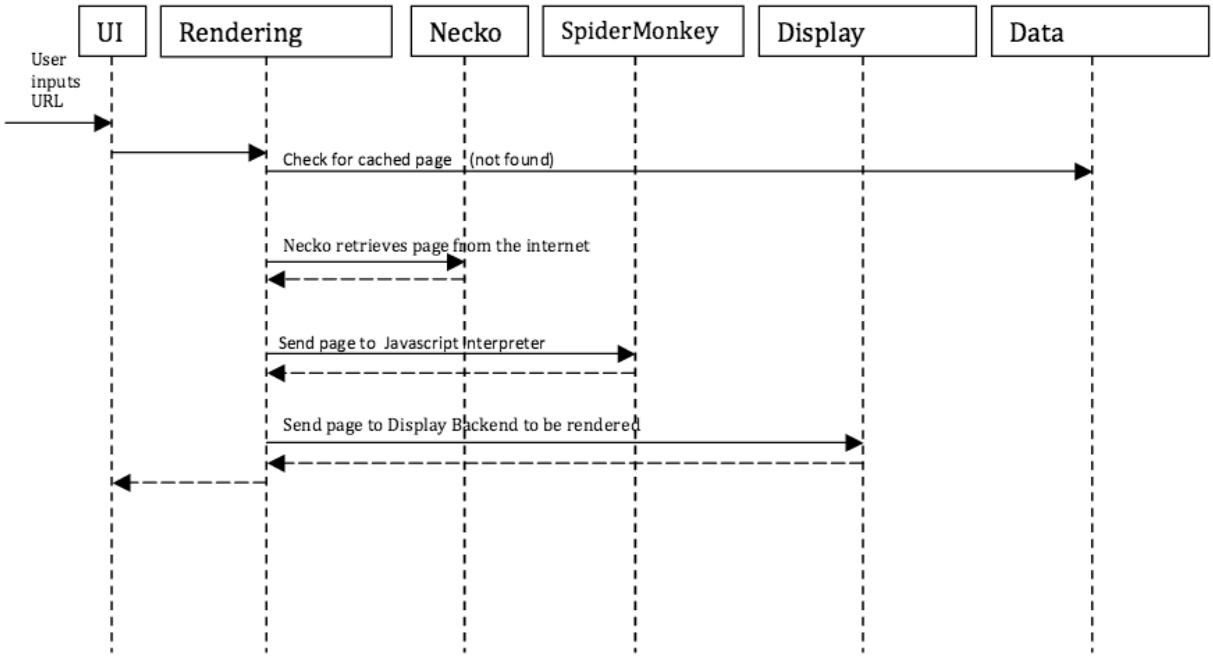
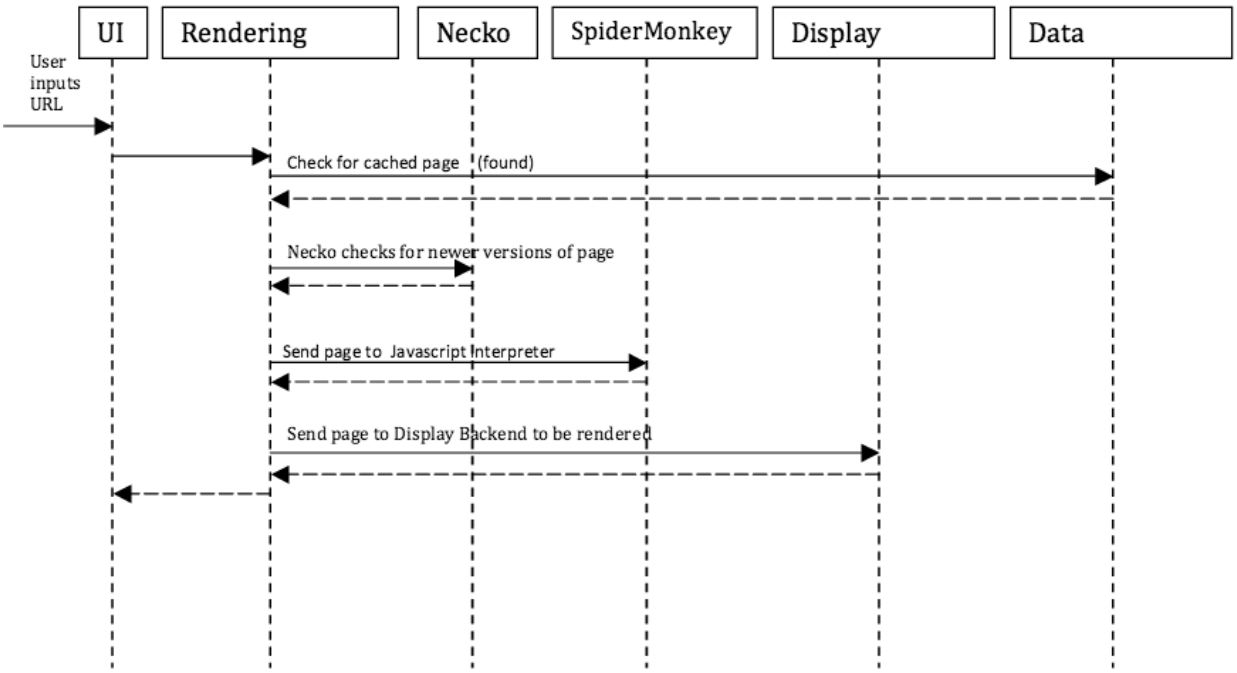


Figure 5.1 Firefox UML Sequence Diagram for Displaying a non-Cached Webpage.

Gecko in order to interpret the page. The page is first sent to SpiderMonkey, Mozilla's JavaScript engine, to interpret and execute any JavaScript. Once this is finished, the page is then sent to the display to analyze the received data, paint it, and send it back to the rendering engine. Gecko then forwards this back to the UI

Firefox UML Sequence Diagram for Displaying a Cached Webpage



The use case in Figure 5.2 shows a UML sequence diagram for a page request that is saved in memory. When a page request comes in from the UI (first layer), an XBL trigger is sent to the Gecko (middle layer) with an event that requests a URL. Gecko checks its database for the webpage, retrieves it and sends a request to Necko to check if the page has been updated since its storage. If an updated version exists, the algorithm explained in Figure 5.1 is executed. Otherwise, the page is loaded from the cache and then the algorithm described for Figure 5.1 is executed.

6. Glossary

COM	Component Object Model
CSS	Cascaded Style Sheets
DOM	Document Object Model
Gecko	Browser Engine, also known as NGLayout
HTML	Hyper Text Markup Language
Necko	Network Library
SpiderMonkey	JavaScript engine
SSL	Secure Socket Layer
TLS	Transport Layer Security
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XBL	XML Binding Language
XML	Extensible Markup Language
XPCOM	Cross Platform COM
XUL	XML User Interface Language
XPCOM	Cross Platform COM

7. Conclusions

Overall, we found that the Mozilla Firefox browser's conceptual architecture closely followed the reference architecture for a modern web browser. It consists of a three-tiered layered architecture. Object-Oriented styles were used on both the top and bottom later, with adapter styles used where needed. The middle layer contains the workhouse of the browser, Gecko, which acts as both the browser engine and the rendering engine by use of a hybrid pipeline-repository architecture.

By centering control on the middle layer, we found that Firefox acts as little more than a wrapper for the Gecko engine. The top layer specifies a graphical user interface and the bottom layer connects Firefox to the outside world, almost everything else is handled explicitly by Gecko. This approach is similar to the reference architecture, which it leverages for the power of separation of concerns. It seems the motivation to do this stems from a desire to allow the Gecko engine to power any web-related piece of

software. By defining clean interfaces between the core of the browser and the components, Gecko is effectively a “browser-in-a-box”.

There was a clear focus on the front-end in the documentation; the pages detailing the functionality of XUL/XBL were the most up-to-date and relevant. Based on the documentation alone, one might think that the bottom components were very stable and static. Some of the XPCOM pages had no major or minor edits in years, sometimes more than a decade. This may be due to an effort to recruit interested developers into working on the project. An unusual flurry of edits occurred on many of the Mozilla Developer Network pages around the middle of 2013.

8. Lessons Learned

This project lead to an increased appreciation and understanding for the need of proper documentation from an external view of a project, as well as the importance of planning in terms of extended functionality and modification.

8.1 Existence of Documentation

Until now, we have worked either on code that was developed personally or in a small group. Having to analyze a large ongoing project with no background led to an increased reliance on documentation and diagrams detailing the architecture and development of the project. This marked a stark contrast relative to the usual initial understanding of one’s own code or the detailed description of ones’ peers.

For documentation to be helpful, it must exist. Much of the architecture of Firefox and its components went undocumented in its early stages and could not be properly analyzed without briefly examining the source code.

This proves to be incredibly difficult for a program on the scale of Firefox and inadvertently creates an entry barrier to any developers looking to join the development. This might also lead to a lack of support for Firefox when software developers are testing their products against modern browsers, or creating extensions/plug-ins/add-ons.

8.2 Updating Documentation

Out-of-date documentation can be worse than no documentation. This can lead the reader to make assumptions that are not true which can cause problems and be potentially dangerous in certain cases. In this case, what documentation did exist on the Mozilla Developer Network was grossly out of date, having not been updated since its initial creation. This means that not only was there information missing in the

documentation, but potentially incorrect information as well. This led us to be skeptical of the entire Firefox documentation due to a possibility that any particular document may be inaccurate.

One example of this was the Necko Networking service provided as part of the Mozilla Application Framework. The documentation regarding this was noted as out-of-date and there was no indication that Necko was still in fact being used, which then had to be confirmed through checking the source code (Ponomarev, Nickolay et al.).

This proves to be tedious and has the same disadvantages and little to no advantage over missing documentation. It can be argued that out-of-date documentation is worse than no documentation as the hours put into the creation or understanding of the documents are now considered wasted and could have been put elsewhere into the project. To add to this, incorrect documentation can, as with no documentation, lead to a barrier of entry to join the project.

In a minimal-effort-attempt to remedy this situation, many of the Mozilla Developer Network pages are actually open to editing by users.

8.3 Planning Future Extendibility

An important factor of any software project is factoring in future extended functionality and maintainability. This was considered throughout the Mozilla Firefox and accounted for in different ways. Some examples include:

- The XPCOM is implemented in such a way that objects can be swapped in or out easily, meaning that Necko or SpiderMonkey could easily be swapped out for alternatives.
- XUL objects follow Object-Oriented style, which allow for easy modification and swapping
- The adapter pattern at the top level allows for many different types of components to connect, which allows extensions/apps to be written in more than just the languages native to the framework
- Gecko uses a pipeline-repository architecture to allow for filters to be easily swapped out, including parsers as well as the content managers such as the Frame Constructor and View Manager

There were some serious oversights when it came to future extendibility.
Due to the in-order parsing done by the compiler, the development of a multi-threaded engine browser proved to be increasingly difficult. This project was put on hold repeatedly, which resulted in Firefox falling behind its competitors in UI responsiveness for heavy web browsing. This resulted in Firefox losing its place as the most popular

browser, since almost every other modern browser became truly multi-processed years ago.

In short, the lack of foresight during the development of the browser engine allowed for quicker development in the short term, but resulted in the project falling behind its competitors in the long term and is now requiring much more investment of time to catch up.

8.4 Use of Gecko

One of the most interesting points we touched on in the report was the use of Gecko. What can essentially be described as a bare-bones browser, it does almost the entirety of the heavy lifting. As we have mentioned before it is used in many Mozilla Framework Applications as the main engine as many of these applications such as Thunderbird or Firefox OS treat content such as web-content, allowing them to make use of Gecko's browser engine/rendering engine functionality.

This has enormous implications as this means that Firefox can be described as more of a wrapper for Gecko that provides both a UI and networking/file-management services. This in turn proves to be advantageous for the Mozilla Foundation as they can invest an incredible amount of effort into Gecko and use it as the platform for both their browsers and operating systems, allowing them to tweak and optimize it much more than competitors would be able to do – consider Apple with Safari and OS X.

Furthermore, this means that applications or plugins developed for Firefox in theory can be used just as easily with Firefox for Android/iOS as well as Thunderbird and Firefox OS. This means that any hours invested into any of their particular applications provides benefits for the entire ecosystem and allows applications to target a much larger market share than the browser itself.

On the other hand, this also proves to have some major drawbacks including supporting various platforms. Due to the nature of Gecko's applications, any changes to Gecko must support running on anything from high-end desktops to low-end phones that run Firefox OS. This manifested itself during the transition to multi-process which is now slated for late 2015 and is not yet available at the time of writing, making Firefox the last modern browser without such support.

Some changes were in fact made to mitigate these disadvantages as the use of XPCOM has been greatly scaled back.

9. References

Angus et al. "Gecko FAQ." 30 September 2015. *Mozilla Developer Network.* Web. 20 October 2015. <https://developer.mozilla.org/en-US/docs/Gecko/FAQ>.

Baron, David and Jonathan Watt. "Gecko: Overview." 1 October 2015. *MozillaWiki.* Internal Wiki. 20 October 2015. <https://wiki.mozilla.org/Gecko:Overview>.

Evans, Ellen et al. "Gecko Embedding Basics." 9 January 2013. *Mozilla Developer Network.* Web. 20 October 2015. <https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/Gecko_Embedding_Basics>.

Fuqiao, Xue et al. "The Joy of XUL." 7 September 2014. *Mozilla Developer Network.* Web. 25 October 2015. <https://developer.mozilla.org/en-US/docs/The_Joy_of_XUL>.

Girad, Benoit. "Off Main Thread Compositing (OMTC) and why it matters." 15 May 2012. *My Programming Experiences.* Web. 20 October 2015. <https://benoitgirard.wordpress.com/2012/05/15/off-main-thread-compositing-omtc-and-why-it-matters/>.

Grosskurth, Alan and Michael W. Godfrey. "Architecture and evolution of the modern web browser." 20 June 2006. *Alan Grosskurth.* PDF. 24 October 2015. <http://grosskurth.ca/papers/browser-archevol-20060619.pdf>.

Hobson, Stephanie et al. "XUL Structure." 21 October 2015. *Mozilla Developer Network.* Web. 23 October 2015. <https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Tutorial/XUL_Structure>.

Ponomarev, Nickolay et al. "Necko Architecture." 8 November 1999. *Mozilla Developer Network.* Web. 25 October 2015. <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Necko/Architecture>.

Tiner, Susan D. et al. "Bird's Eye View of the Mozilla Framework." 23 November 2015. *Mozilla Developer Network.* Web. 20 October 2015. <https://developer.mozilla.org/en/docs/Bird's_Eye_View_of_the_Mozilla_Framework>.

Waterson, Chris et al. "Introduction to Layout in Mozilla." 10 June 2002. *Mozilla Developer Network.* Web. 20 October 2015. <https://web.archive.org/web/20150919100442/https://developer.mozilla.org/en-US/docs/Introduction_to_Layout_in_Mozilla>.

Young, Alex R. "Multiprocess Firefox." 5 December 2013. *DailyJS.* Web. 25 October 2015. <http://dailyjs.com/2013/12/05/multiprocess-firefox/>.