Concrete Architecture of Firefox 6
Team Phoenix

Mark Simon - 0ms16@queensu.ca
Colin Murdoch - 7cm53@queensu.ca
Boris Madzar - boris.madzar@queensu.ca
Matt Jewett - 7mcj@queensu.ca
Michael Bertoia - 4mb3@queensu.ca

November, 2011

List of Figures

Contents

Abstract

In this report, we will discuss the concrete architecture of the Firefox web browser, with specific focus on the user interface and data persistence components. We begin with a discussion of the process we used to derive the concrete architecture, as well as the individual components and dependencies between the UI and data persistence components, and the dependencies between those and other components in Firefox. We determine that the concrete architecture of Firefox is a weak layered architecture, with four extremely interconnected layers: the UI, browser engine, rendering engine, and backend components, as well as XPCOM, which is used for communication between the layers, and libraries and build support. Then we discuss two use cases: building the user interface, and accessing the application cache.

1 Introduction

This report is concerned with the Concrete Architecture of the user interface, and data persistence subsystems of Firefox 6.0. It also includes a derivation of the overall concrete architecture of Firefox 6.

Using LSEdit, the landscape file provided to us, the Firefox source code repository, as well as the actual source code of Firefox 6, we divided the source code files into the various subcomponents that make up Firefox. We then properly distributed these subcomponents amongst the main architectural components of Firefox identified in our original conceptual architecture.

Further examination of the source code and dependencies warranted some changes to our overall architecture. Three additional components were defined: build support, low-level Support, and libraries. Furthermore, the XUL parser, originally considered a separate component in our conceptual architecture, was redefined as part of the toolkit API. Similarly, XPCOM was defined as part of low-level support.

Ultimately, the overall concrete architecture of Firefox 6 was determined to follow a layered style, albeit a weak one with many exceptions. The user interface subsystem was also found to have a layered architecture with two layers, the Firefox UI and toolkit API. The data persistence Subsystem didn't follow any architectural style discussed in this course, but was instead a collection of four largely independent subcomponents.

Derivation Process We were initially provided a landscape file containing all the Firefox source code files, with their dependencies to one another. We consulted the Firefox source code directory structure website, as well as some additional write-ups on specific sub components that it linked to. Using this information, as well as an examination of actual source code when needed (to ascertain any information not provided on this website), we divided the aforementioned files into their appropriate subcomponents. Furthermore, as neither had any bearing on the actual implementation of Firefox, we removed all testing and sample code from this set of files. Specifically, the sample code was in fact creating some dependencies in our LSEdit representation that weren't actually present in the Firefox implementation.

Using our original conceptual architecture as a starting point, we initially attempted to distribute these subcomponents amongst the main components of Firefox's overall architecture, however new information discovered during the derivation process warranted some changes to this original architecture. Specifically, three new components were added to this architecture, and two existing components were re-defined as subcomponents of other components.

We added the build support, low level support, and libraries components to our overall architecture to categorize subcomponents and files that didn't logically belong in any of the existing components.

After further examination of the function of XUL, we revised our original placement of it in our overall architecture, and now consider it a subcomponent of the toolkit API within the UI subsystem. Furthermore, since it performed functions comparable to other subcomponents of the low level support component, XPCOM is now considered a subcomponent of low level support.

After we settled on an initial concrete architecture, with all files placed into their appropriate subcomponents, and all subcomponents arranged into their respective high-level components, we continually double checked and updated this architecture to reflect any newly discovered information.

Considerably more effort was spent on the derivation of the UI and data persistence subcomponents, as these were the two sections we were tasked with analyzing. Effort was taken to confirm the respective architectures of these subsystems, making sure that certain subcomponents didn't belong in other subsystems, and that subcomponents from other subsystems didn't belong in either of these components.

Once we were sufficiently confident with the architecture of each of these subsystems, we meticulously examined the source code of all files contained in each subsystem, confirming that dependencies were actually justified by the code, and getting a better sense of function and data flow between components.

2 Architecture

Overall System Architecture In order to properly develop the architectures inside the UI and data persistence modules, we needed to model the entire system to understand the connections to various sub components. This also allowed for a better reflexion analysis because the entire system could be contrasted against the conceptual architecture, rather than singular sub-components. Seeing all the dependencies in the

system at a very high level allowed easy isolation of divergent dependencies - both expected and unexpected. However, it should be noted this extra effort was needed to get a better understanding of the UI and data persistence sub-systems.

Once the system was organized, we found the overall architecture demonstrated two different styles: when browsing through source code, it seemed that the developers had an object-oriented architecture in mind to develop the system; when looking at the entire system and its dependencies with LSEdit, the system demonstrated a weak layered style comparable to our conceptual architecture predictions.

The object-oriented architecture was reasonable as the nature of a component such as XPCOM is to split the system into several individual components that could be thought of as objects; if a new piece of code wanted to interact with another component, it would directly call it, rather than going through a layered hierarchy.

The layered architecture was expected to be seen and we feel this conveys the overall architecture of the system. Looking at the LSEdit diagram (Figure 1), the dependencies displayed suggest a weak layered architecture. Each component appears to depend more on the expected components from the conceptual architecture rather than the observed divergences - many of the strong dependencies displayed fit into a layered architectural style. Due to the very large amount of dependencies there are plenty of exceptions which is why we feel this is a weak layered architectural style.

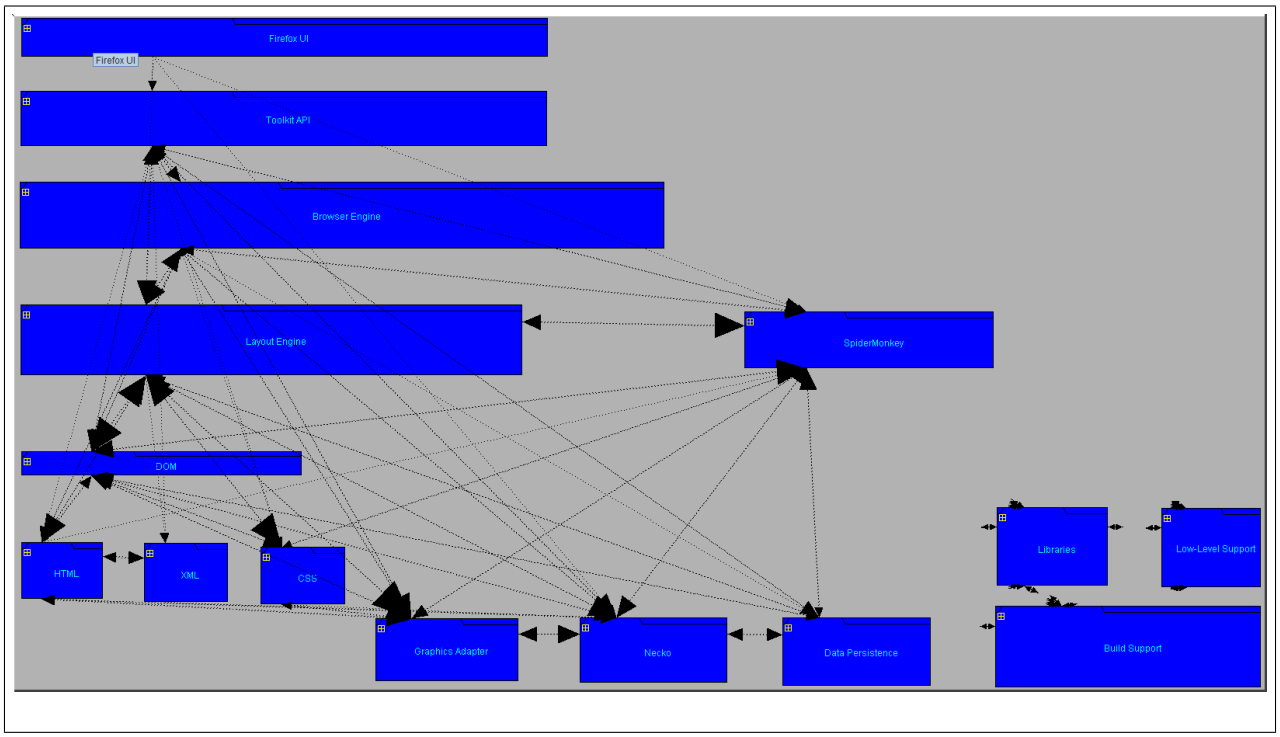

Figure 1: LSEdit view of the architecture

In addition to looking at the overall architecture, we also examined several different design patterns that were displayed in the UI and data persistence modules. Of note, we found appearances of the iterator, observer, façade and singleton patterns. The façade pattern was displayed in data persistence with mozStorage and DOM storage, with both elements acting as a façade to SQLite3. There was also an instance of singleton pattern in data persistence with the startup cache component. There was some evidence of the toolkit API component using iterators and observers in its files.

Conceptual Architecture vs. Concrete Architecture

The concrete architecture mapped fairly well to the conceptual architecture. The conceptual architecture derived in the first phase of the project is shown in (Figure 2). The Visio representation of the concrete

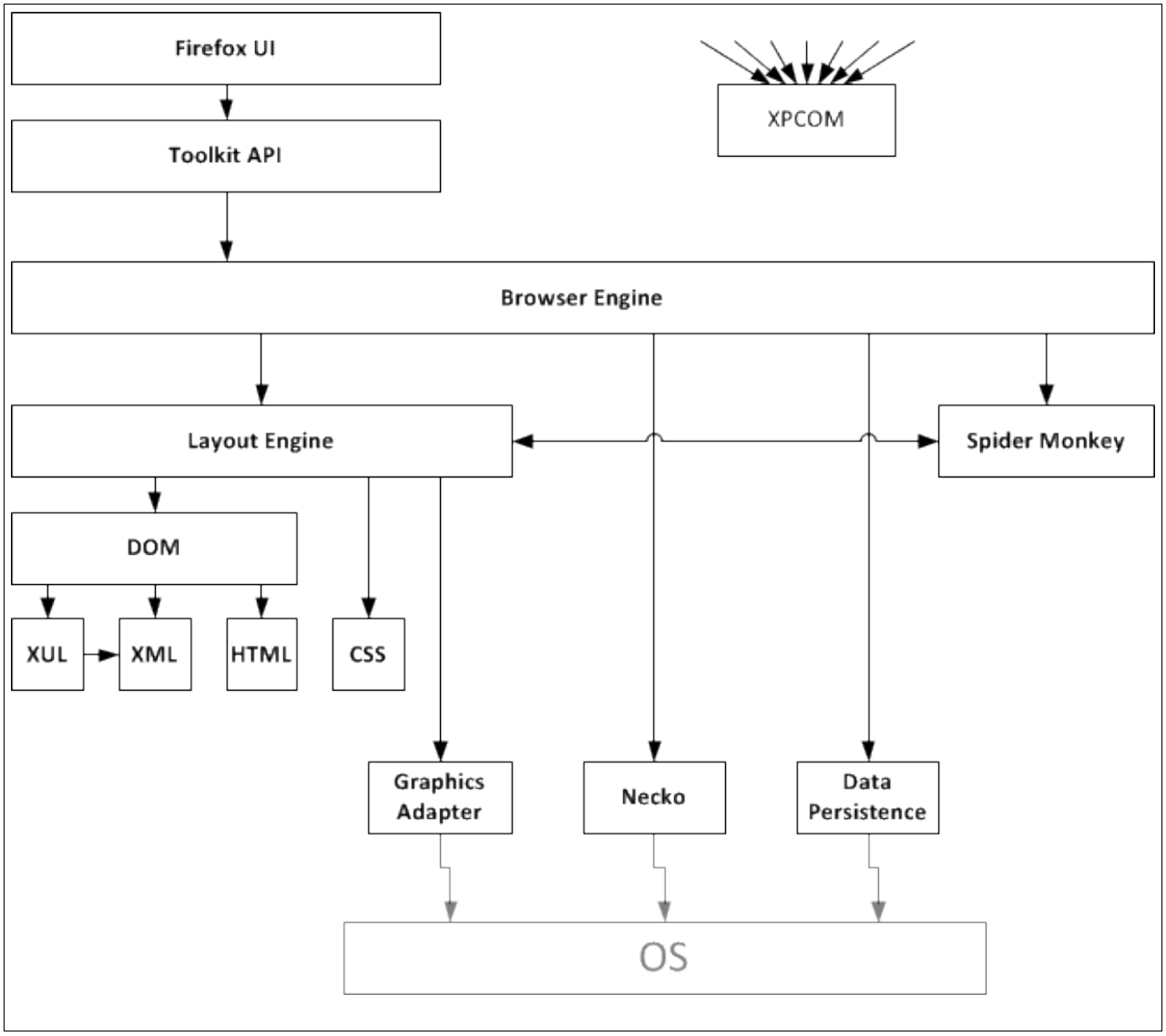

Figure 2: Conceptual architecture of Firefox

architecture is shown in (Figure 3).

Looking at the two diagrams it is evident that not every component transferred, and some new ones were created. In particular, we found that the XUL parser was no longer servicing the rendering engine and instead it was merged into the toolkit API. Besides this, we created three abstract layers similar to XPCOM in the conceptual architecture - components that serviced the entire system and were either called upon by everything or used to build Firefox. These new components are build support, low-level support, and libraries.

Build support was a collection of tools used throughout the entire system and was generally used for installing Firefox and building the system. Low-level support contained XPCOM and other similar files that allow for the entire system to interact. These were not considered in the conceptual architecture. Finally, libraries contained many different library structures that many components referenced in order to properly function.

Discrepancies and Issues When the system had been fully organized in LSEdit, it appeared there were 85 out of 132 possible connections, not including the abstract components such as low-level support or libraries. This is significantly higher than the prediction in our conceptual architecture which only had 14 connections.

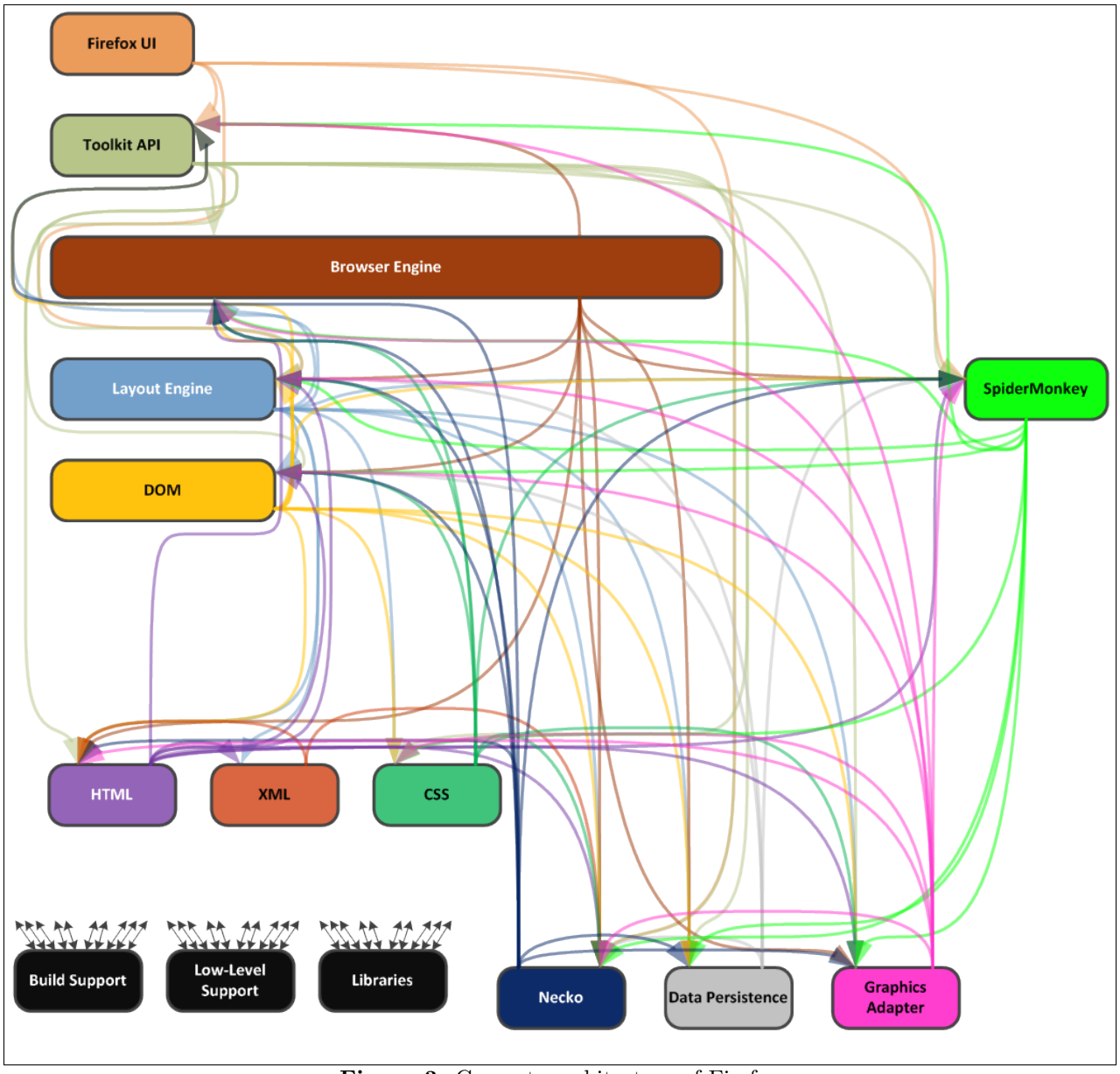

Figure 3: Concrete architecture of Firefox

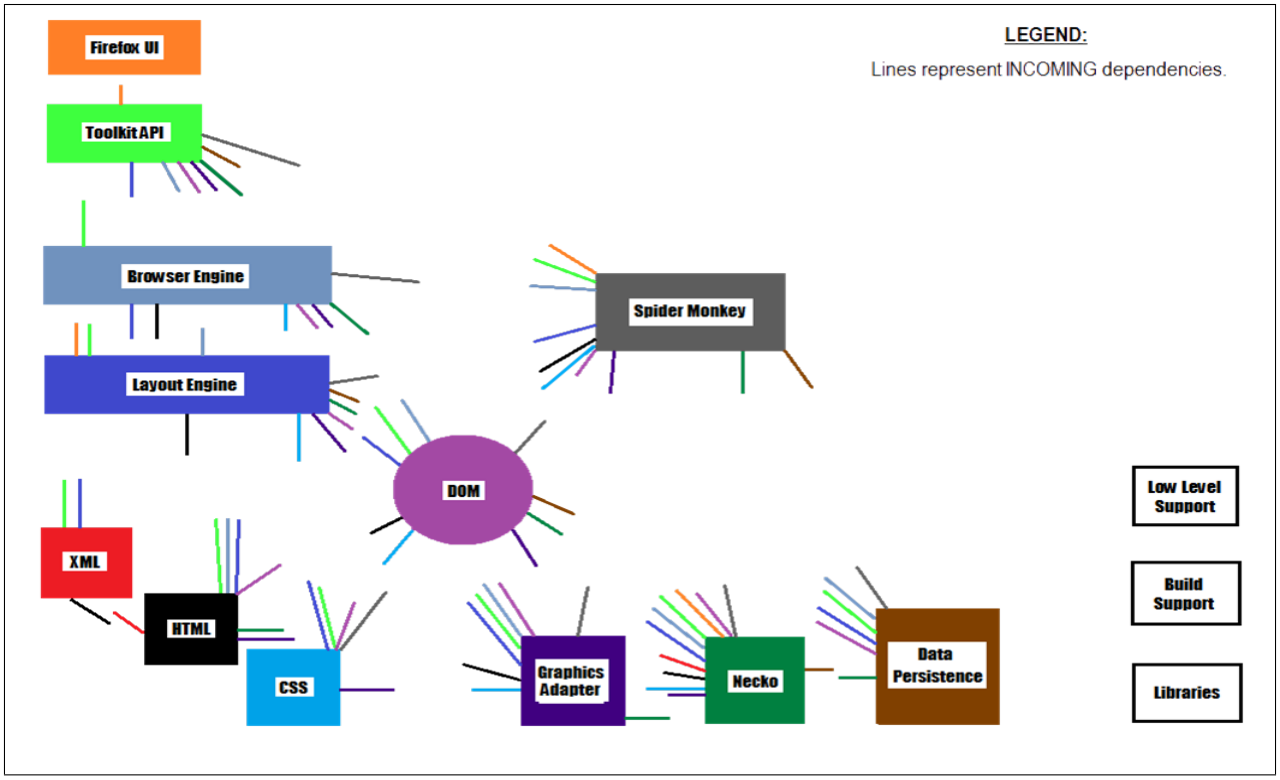

Figure 4: A simplified version of the concrete architecture

However, when we examined the source code we determined that a lot of code developed by third-parties included copies of common libraries which were redundant. After going through the LSEdit file and removing redundant dependencies, we concluded that there were only 83 out of 132 possible connections. This process of removing redundant references did not remove many existing dependencies entirely, but it did cut down on the amount of dependencies between subsystems by quite a lot.

Some interesting dependencies were also examined in the UI and data persistence modules and we mention a few of these unexpected but explainable dependencies later in the report. A brief overview of these include: toolkit depending on HTML parser, chrome depending on Necko, mozStorage depending on layout engine, DOM storage depending on layout engine, mozStorage depending on Necko, and DOM storage depending on Necko.

Finally, it was noticed that the original landscape file did not include all file-types, but rather only C source files. While this made the system much more manageable and was a justifiable choice, it did make the UI system harder to visualize since it is primarily implemented using other languages. With this in mind, it would be expected to see more dependencies to or from the UI component if more file-types were included.

Figure 4 As it tends to happen in diagrams with a sufficiently large amount of dependencies, (Figure 3) and (Figure 1) are difficult to visually comprehend. The high volume of arrows, and the fact that they converge on one location on each box, causes many lines to overlap and crisscross. It is very difficult to determine from these diagrams how many components depend on any other component, and conversely, how many that component depends on. Even with the colour-coding used in (Figure 3), overlapping lines sometimes cause confusion as to where an arrow came from, especially if the latter portion of the arrow is visually obscured by other arrows. Relying solely on these diagrams for information about dependencies can be very time consuming, and there is a risk of wrongly interpreting the information presented.

To address all these issues, we constructed another diagram using abbreviated arrows (Figure 4). Arrows of a certain colour correspond to the component of the same colour. The only arrows pictured are incoming dependencies: for example, since the XML parser (coloured red) depends on Necko, there is a short red arrow pointing at Necko, located on Necko. Bidirectional dependencies have no unique visual representation: they

are simply represented by having one arrow on each of the two components involved.

In this diagram, arrows are not constrained to one location on the surface of a box. This prevents any overlapping or hidden arrows. Arrows are also oriented such that they are located directly between the depending and depended-on components, aimed back at their source. This makes it easy to find arrows, as there is no mystery to where the arrow will be, if it exists at all. For example, Necko depends on the Data Persistence Subsystem. Since Necko (green) is located to the left of Data Persistence, there is a green arrow located on the left side of the Data Persistence box, positioned at the same height as the middle of the Necko box, and aimed directly back at the Necko box. All arrows in the diagram are positioned following the same conventions.

Counting the number of incoming dependencies is very easy with this diagram.

3 User Interface Subsystem

Overview

To match the proposed conceptual architecture, the user interface was divided into two separate, high-level subsystems: Firefox UI and toolkit API. The Firefox UI subsystem consists of Firefox-specific code and UI elements, while the toolkit API subsystem contains code common to many Mozilla and XULRunner-based applications.

The conceptual architecture predicts that these two subsystems will form a layered architecture, with the Firefox-specific code constituting the top layer and the toolkit API constituting the bottom layer. The concrete architecture does show this to be the case, however there is some crossing of layers introduced where the top layer makes calls down to low-level components directly. This is to be expected in the actual implementation of a layered architecture, and does not indicate that the original conceptual architecture was incorrect.

XPCOM and Dependencies Most communication between subcomponents in Firefox is done through XPCOM, Mozilla's cross-platform component object model. To communicate with a component registered with XPCOM, you require one of two identifiers for the target component: its class ID (CID), a 16-byte integer, or its contract ID, a string. The Mozilla developers package all the CIDs and contract IDs for a given component as macros in a C++ header file that resides in the build directory for that component (e.g. nsWidgetCID.h and nsBrowserCompsCID.h for /mozilla/widget and /mozilla/browser/components, respectively). This expedites development and improves code readability by eliminating magic numbers from the code.

These CID files both help and hurt our ability to identify dependencies between components: on one hand, they allow us to identify that a dependency exists between two components since the calling component is including the CID definition file that resides within the called component; on the other hand, by including just the CID file we cannot easily pinpoint why the dependency exists. The only way to determine which functions and services of the called component the caller is using is by examining the source code line-by-line, which there was not enough time to do. As such, we could not inspect all the dependencies in detail, and our understanding of the concrete architecture may not be as complete as it should be.

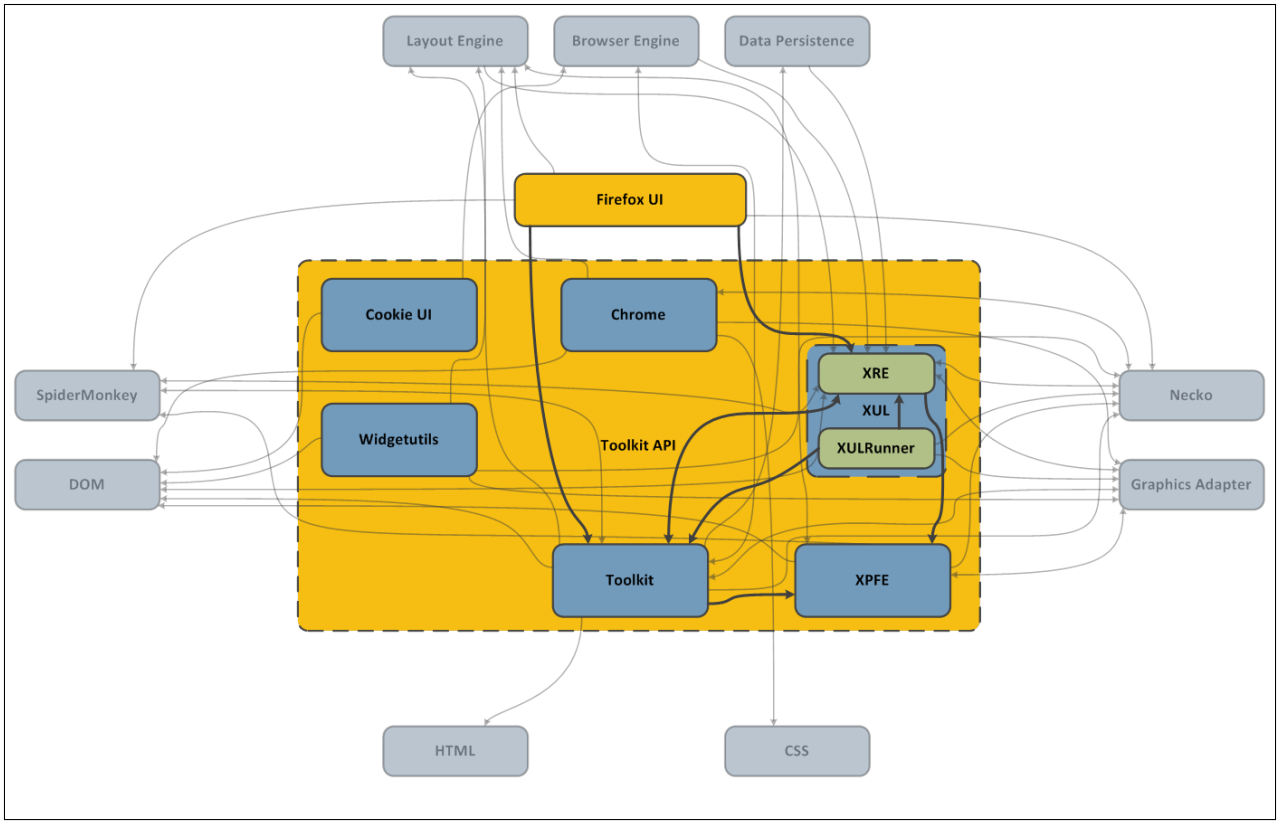

Figure 5: Architecture of the user interface component

Firefox UI

The Firefox UI subsystem(Figure 5) consists of the /mozilla/browser subdirectory and contains all of the Firefox-specific UI components as well as some code responsible for managing Firefox's overall operation. If we consider dependencies alone, some of the code from this latter group would have fit better in the browser engine; however, it consists mostly of C++ wrappers for XUL components and interacts very closely with the XUL-based UI. For this reason it was decided that conceptually it belongs in the UI subsystem and any calls to lower-level components should be treated as divergences from the conceptual architecture.

The majority of the files that compose this subsystem are XUL files - that is, written in XML and JavaScript - and resources such as images. The importance of this subsystem is therefore greatly underrepresented by the LSEdit dependency graph since it only considers C/C++ source code.

Dependencies The Firefox UI subsystem is the entry point for Firefox, and belongs on the topmost layer. nsBrowserApp, which is defined here, is the component which actually bootstraps XULRunner to load the UI and does the processing of command-line parameters to load the application. Firefox UI is therefore the only subsystem with no incoming dependencies.

The C++ wrapper code communicates with the rest of the system through XPCOM (for the most part), and therefore the only internal dependencies are between the individual wrappers and the common browser CID definition header. This is done for expediency in development - rather than having to include a CID header for each wrapper component, a developer can simply include nsBrowserCompsCID.h and get all the XPCOM registration information they need.

Outgoing dependencies are to the toolkit subcomponent of toolkit API, which is needed to load the user profile as specified on the command line, and to the XUL subcomponent which actually reads the XUL definition files and creates the application UI. There are dependencies out from the wrapper code to Necko and SpiderMonkey to provide common utility functions (such as nsNetUtil for URI parsing support) as well

as to identify MIME types that need special UI support, such as RSS feeds.

Toolkit API

The toolkit API subsystem is not as clearly defined in the Firefox source code as the Firefox UI subsystem. It is comprised of six individual components with differing levels of interconnectivity, but with the common property of being responsible for creating the user interface and providing UI services to other subsystems.

Some dependencies into and out of the toolkit API are not true dependencies, but rather dependencies created due to files and functions with the same name existing within system libraries and the Mozilla source code. One example of a false dependency is the outgoing dependencies many components have to a file called unistd.h in a third-party library within the graphics backend. However, unistd.h is a common system C library, and all of the dependencies are in fact to the system library and not to the one inexplicably included in the graphics backend. There were several other similar cases, but none of them were as widespread as the unistd.h dependencies.

Some of the other dependencies were true dependencies but were not meaningful in terms of determining the architecture. For instance, many components depend on nsAppRunner (actually, nsXULAppAPI.h is the file included in the source code, but this file was not to be found). On further examination, it was determined that the sole function that is used by the code that depends on nsAppRunner is XRE_GetProcessType(), a function that determines what type of process the current instance is - chrome or content. This is mainly used in assertions, since a content process is much more restricted than a chrome process and is not permitted to make certain calls and access certain data. One explanation for why this dependency exists is that this is relatively new functionality that may have been wedged in where necessary, and has not yet been extracted to a common library. In terms of the architecture, we are treating these calls as calls to a common library and not considering them in the architecture analysis.

We did not really consider the subcomponents of toolkit API in our conceptual architecture, and as such we did not expect any particular architecture. In the end, it seems that there is no discernible architecture within the toolkit API component; it consists of three mostly independent subcomponents (cookie UI, WidgetUtils and chrome) and three inter-related components (XUL, toolkit and XPFE). This is understandable since the toolkit API is an artificial subsystem we created based on our conceptual architecture, unlike many of the others it does not actually exist in the source code.

Cookie UI and WidgetUtils These are very small components that have been extracted from the extensions directory - cookie UI is responsible for the cookie permissions UI and WidgetUtils is responsible for mouse scrolling support. WidgetUtils could have been placed with widget in the graphics adapter; however mouse scrolling is higher-level functionality than widget painting so it belongs with XUL rather than widget.

Dependencies are only outgoing, with cookie UI depending on the cookie permissions code in the browser engine and the window creation functionality within DOM to support the popup window component. WidgetUtils depends on event listeners in the graphics adapter, layout engine and DOM as well as on common utility functions found in Necko.

XUL (XRE and XULRunner) The XUL subsystem consists of the XRE component from the toolkit directory and the entirety of XULRunner. XULRunner is meant to replace XRE, but this is an ongoing process and much of the low-level code has not been migrated yet. The XUL subsystem is responsible for loading and parsing XUL files and generating the UI.

Incoming dependencies are mainly from the Firefox UI to bootstrap the application, and from toolkit since the two subsystems are interdependent - XUL contains supporting code that toolkit needs, and vice versa. Other incoming dependencies are for certain very specific uses, such as Windows DLL support. Outgoing dependencies are on the graphics adapter for creating widgets, DOM for creating and managing windows, and SpiderMonkey for parsing JavaScript. Internal dependencies are to and from toolkit, mainly for loading user profiles from data persistence and to XPFE which contains additional code for managing and creating windows. We could not find a good explanation as to why some window management code was in DOM and

some was in XPFE, it seems to be a holdover from the days when XPFE managed the UI.

Chrome The chrome subsystem implements the chrome registry, a common service that all chrome providers (such as the browser UI, extensions, and other chrome packages) must register with. It is responsible for maintaining this registry, and responding to requests for information as to where chrome packages are stored on disk. It is also responsible for coordinating window repainting when chrome changes.

There is only one incoming dependency to chrome: from Necko, used to include utility functions for the chrome protocol handler. Outgoing dependencies are to the layout engine, graphics adapter and DOM to aid in repainting of the chrome, to Necko to provide chrome protocol handling and to the CSS parser to load style sheets included in chrome packages. The sole internal dependency is to XUL, which is necessary as XUL is the primary chrome provider in Firefox.

Toolkit Toolkit provides a set of UI-related utilities, such as the crash reporter, updater and common windows and tools (e.g. file picker, parental controls, bookmarks, etc.). In terms of the source code directory structure, XRE actually is a subsystem of toolkit; however it was grouped with XULRunner to create a XUL subsystem which seemed more logical.

Due to the fact that it is a collection of loosely-related components, toolkit has dependencies to many other subsystems. A few that were interesting and unexpected include a dependency from toolkit to data persistence to provide user profile storage and a dependency on the HTML parser for reading and writing the bookmarks.html file which stores the bookmarks separately from the mozStorage database. There is a dependency coming in from the top level UI to be able to load a profile from the command line, and there are strong bidirectional dependencies to the XUL subsystem as the two work in tandem to create the UI.

XPFE Most of the code that used to make up XPFE has been migrated to the browser component, which is in the Firefox UI subsystem. All that remains in XPFE is the low-level window creation and management functions. Most of the dependencies coming in are to nsAppShellCID, the XPCOM registry data for the application shell service, the topmost component in the UI management hierarchy. There are bidirectional dependencies with the layout engine and the graphics adapter, since all three components are responsible for part of the layout/painting process. There are also smaller dependencies out to Necko, DOM and SpiderMonkey for providing common utilities and support services.

4 Data Persistence Subsystem

Overview

The data persistence component of Firefox (Figure 6) is responsible for storing and retrieving data. It is divided into four distinct subsystems: the DOM storage, mozStorage, profile, and the startup cache.

In addition, there is code for a database backend called Mork, which was developed by Mozilla for the purpose of storing persistent data. As of Firefox 7, Mork will have been completely phased out, and isn't really used in recent versions. For this reason, it will not be discussed in detail here.

The data persistence component itself does not have any particular architectural style; it is more a collection of distinct modules. However, the components themselves are laid out in particular styles. The mozStorage element is a layered architecture that is built using a façade design pattern. The DOM storage element is very similar in design.

mozStorage

mozStorage is a database interface to the database backend, which is implemented as an SQLite3 database. SQLite3 is a server-less version of a relational database. In order to maintain modularity, the façade design pattern is used, with mozStorage acting as a façade on top of the backend, which essentially allows for the

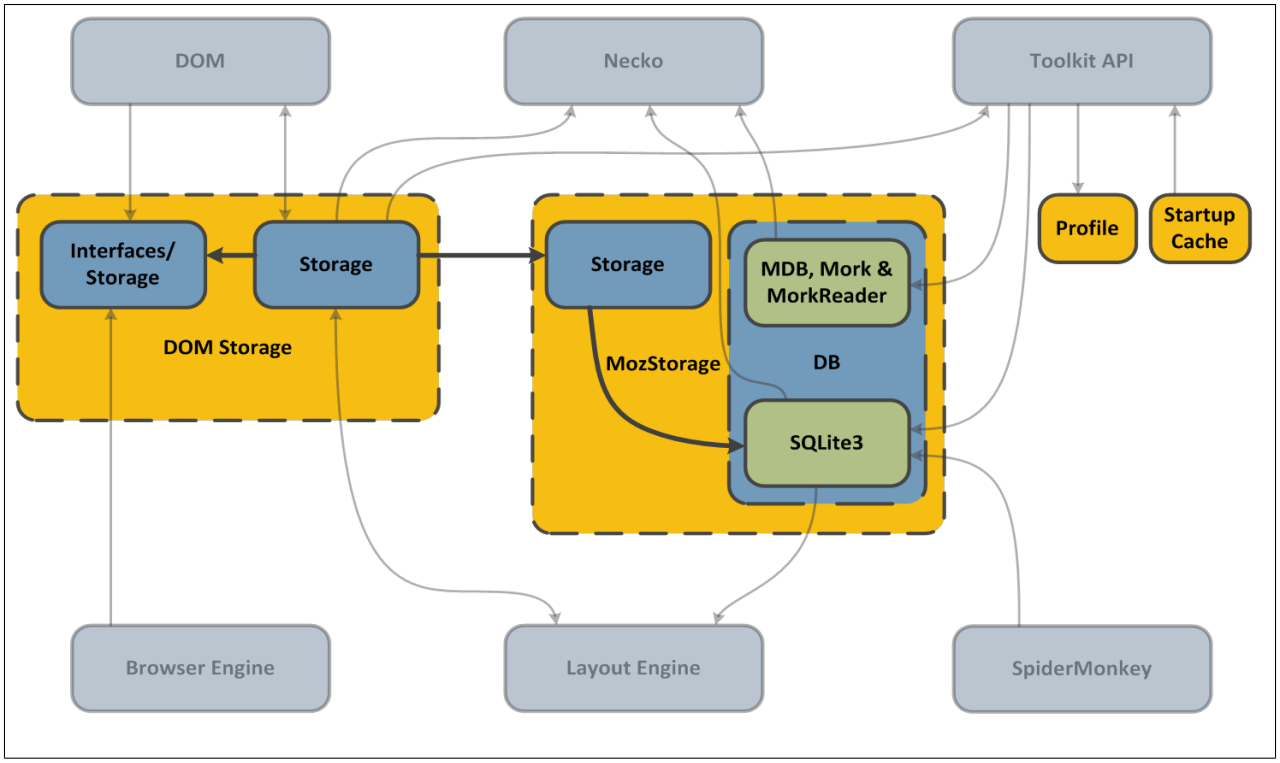

Figure 6: Architecture of the data persistence component

backend to be implemented using a completely different system. In this case, the façade would need to be changed to accommodate the new backend, but to a developer wishing to use the database component, the interface would remain the same.

mozStorage contains three main interfaces for interacting with the SQLite3 database: mozIStorageService, mozIStorageConnection, and mozIStorageStatement. mozIStorageService is the main component, and handles the creation of connections into the database backend. It is also used for creating backup copies of unopened database files. mozIStorageConnection is the main interface used for interacting with the database; it provides methods for testing connections, executing simple statements, and handling errors. mozIStorageStatement is an abstraction of a prepared statement, to which you can bind parameters and handle the results from the execution of the statement. Other interfaces include mozIStorageValueArray, which is a wrapper for values returned by the execution of statements, and mozIStorageProgressHandler, which monitors the progress of a statement execution.

The SQLite3 database has incoming dependencies from the mozStorage interface, as well as from SpiderMonkey, which is caused by some exception handling code, and the toolkit API, caused by the need for the UI components to display things like history, bookmarks, etc. In addition, there is a dependency from mozStorage to the actual SQLite3 backend, which is self-explanatory since the interface calls functions within SQLite3 itself. No outgoing dependencies exist, due to the fact that it is a third-party library, and can't possibly have any real outgoing dependencies. There is a flase dependency going to Necko, caused by the fact that there is actually a duplicate of the SQLite3 code within Necko.

DOM Storage

The DOM storage component is the implementation of a W3C specification for storing session information on the client side. DOM storage is similar to cookies in that it offers web developers a means of storing information in the form of key/value pairs, but there are a few key differences. For instance, DOM storage does not transmit values to the server with every request as cookies do, nor does the data in the local storage area ever expire. In addition, the space allocated for DOM storage is much larger than a cookie.

The dependencies between DOM storage and the DOM are there because there is some DOM storage

permissions code in the DOM component that fit better there, but in actuality is a part of the DOM storage system. Most of the dependencies exist for similar reasons, with the exception of the dependency to the layout engine, which exists because there is a file called nsContentUtils.h, wherein there is a class definition with the same name that is used by the DOM storage for determining access levels for other functions. The browser engine depends on the DOM storage interfaces for accessing the DOM storage to determine if information requested by a page is located there. Dependencies to Necko exist because the objects in the DOM storage need to be able to communicate with the server.

There is an internal dependency from DOM storage to mozStorage, which exists because DOM storage needs some functionality provided by mozStorage for storing persistent session data on the client machine.

Startup Cache The startup cache is used for making the browser load quicker. It was implemented as a replacement for fastload, which served the same purpose. When the browser is initialized, the information that is in the startup cache is loaded. It is implemented using a singleton design pattern; there is only one instance of the cache loading object at any given time, and it is usually only accessed during startup.

Profile The profile object is responsible for accessing profile information on the filesystem. When a profile is loaded, the toolkit API calls the profile interface, which gets the information that is required and passes it back to the toolkit API for displaying the actual profile.

5 Use Cases

The following are two use cases which we analyzed based on our concrete architecture. Since we only focused our concrete architecture analysis on the UI and Data Persistence components of Firefox, the steps within our sequence diagrams which do not involve these two components are based on our conceptual architecture.

One of the main objectives of this report is to perform a reflexion analysis in order to compare our conceptual and concrete architectures. To that end, the two use cases we have selected incorporate divergences we found when we extracted the concrete architecture of Firefox. In order to highlight the rationale behind these divergences we will briefly explain which features of Firefox each divergence supports. In order to determine which features are responsible for the unexpected dependencies, we analyzed the Firefox source code by looking at comments, C++ function and class names, as well as the Mozilla Developer Network documentation for clues as to why they were introduced.

Use Case 1: Building the UI at Startup (Figure 7)

When we extracted our conceptual architecture for Firefox one of the details we were not clear on was the processing of the user's profile. For example, Firefox users can download "personas" which are images that are used to personalize the browser chrome component of the UI, which surrounds the content area of the browser. After extracting the concrete architecture for Firefox, we analyzed the source code and discovered a divergence that accounted for the step of loading the user's profile. The toolkit API, using the profile subcomponent within data persistence, loads the user's profile at startup. This use case demonstrates the toolkit API to data persistence dependency.

Step 1: The user starts the Firefox web browser.
Step 2: The toolkit API loads the user profile via a request to data persistence.
Step 3: Control returns to the toolkit API which calls the layout engine to initiate the build process for the browser chrome component of the UI.
Step 4: The layout engine makes use of image libraries to load the user's "persona" if one is installed.
Step 5: Finally, the layout engine calls the graphics adapter to paint the screen.
Note: At this point the browser will load the user's homepage, and perform other startup procedures - these have been excluded for brevity.

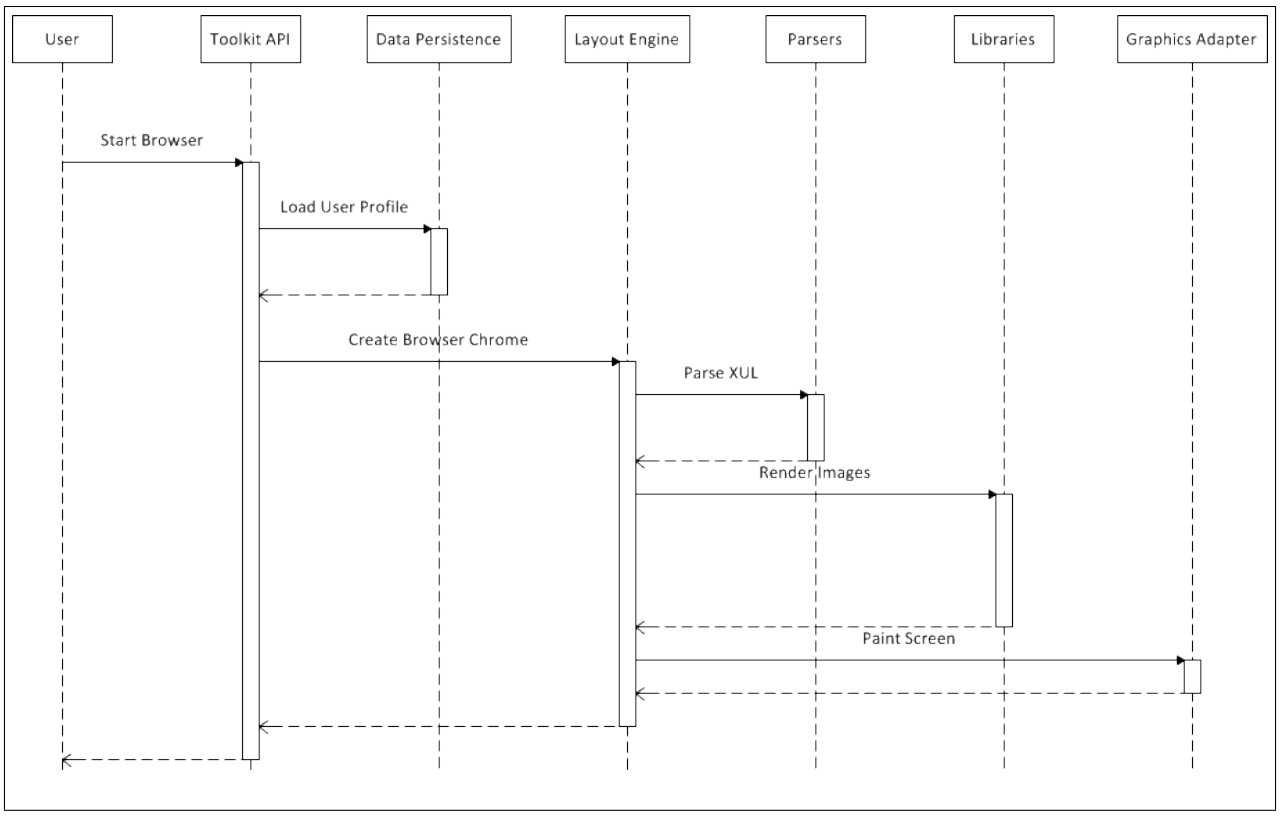

Figure 7: Building the Firefox UI

Use Case 2: Accessing the Application Cache (Figure 8)

One of the features supported by HTML5 is the ability to store web application data in a persistent location known as the application cache. The HTML5 specification includes the manifest attribute which specifies a file that lists certain web application resources which should be stored for future use. When a user accesses the web application again the resources are loaded from the application cache rather than being re-downloaded. Only when the manifest file's contents have changed are the resources downloaded again. Thus, in order to implement this functionality a dependency was created from Necko to data persistence so that resources can be stored in, and obtained from, the application cache. This dependency was not present in our conceptual architecture.

For the following use case we assume that the user has already visited a web application and has stored some resources in the application cache.

Step 1: The user enters a URI to a web application in the address bar.
Step 2: The URI is passed through the toolkit API to the browser engine.
Step 3: The URI is passed from the browser engine to Necko.
Step 4: If the manifest attribute is present Necko makes a call to data persistence in order to load the stored resources.
Step 5: Control is returned to the browser engine and makes use of the parsers to display the web application.

Note: The final step of painting the screen would require the graphics adapter, but this has been excluded to save space. Also, in order for Necko to determine that the manifest attribute has been specified we expect that there should be a dependency from Necko to the HTML5 parser. However, we did not study these components of Firefox so we have not included this step in the sequence diagram.

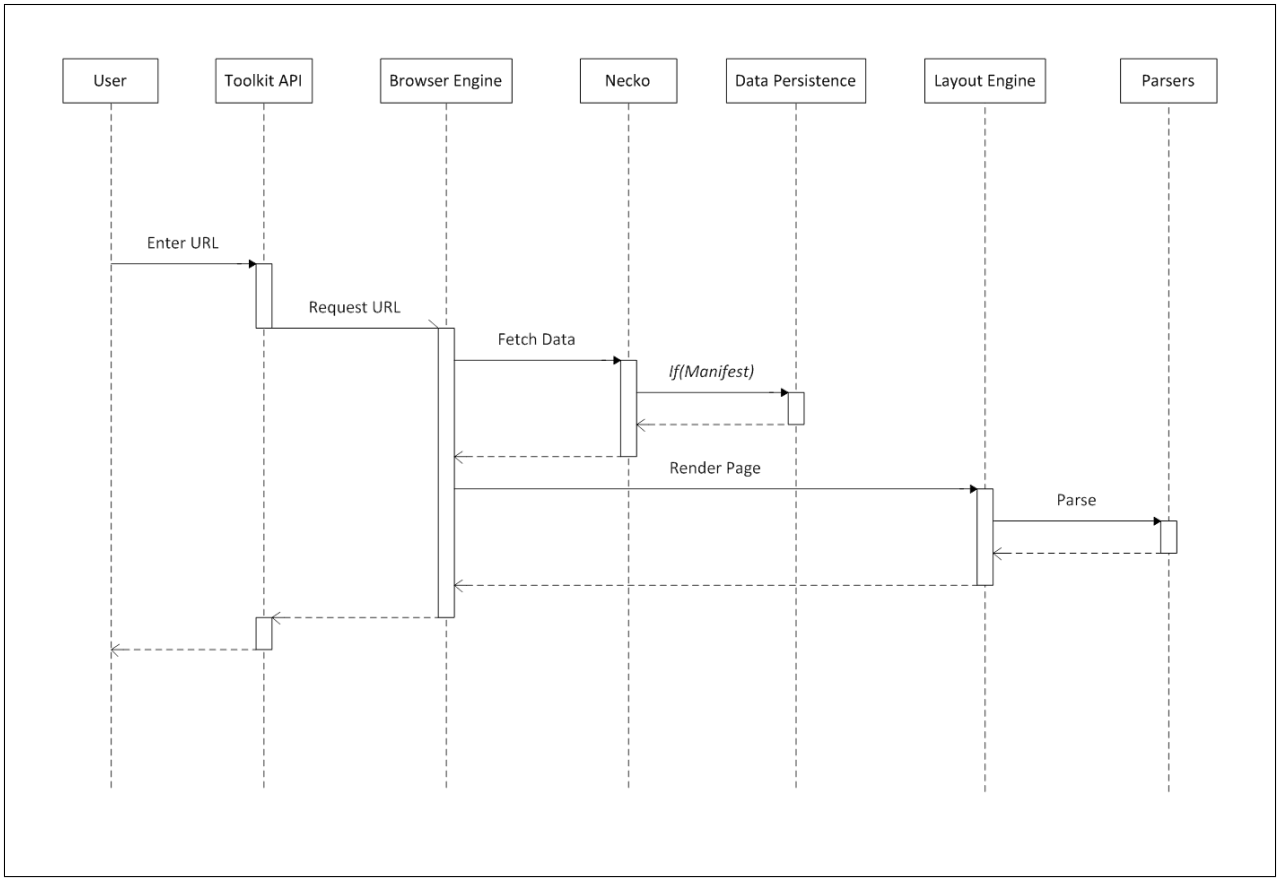

Figure 8: Accessing the application cache

6 Conclusions

Our research into the concrete architecture of Firefox 6 ultimately supported our conclusions about the conceptual architecture: the overall concrete architecture of Firefox 6 is a layered architecture, albeit a weak one with lots of exceptions and cross-level dependencies. In fact, at the file level, Firefox has many characteristics consistent with an object-oriented architecture.

The data persistence subsystem of Firefox doesn't follow any traditional architectural style discussed in the course. It is essentially comprised of four largely independent subcomponents that interact primarily with other components outside of data persistence.
The user interface subsystem has a layered architecture, with two layers. This is what we expected to find, and is consistent with our conceptual architecture.

Lessons Learned Our derivation process yielded more dependencies than we expected, including some that didn't initially seem logical. Upon further investigation, virtually all of these surprising dependencies turned out to be valid. This experience taught us that despite how simple and logical a program can be theoretically (i.e. its conceptual architecture), the interconnections in sufficiently complex programs tend to, in reality, be much more diffuse and numerous.

Many other groups discussed the difficulty in working independently on this project. The primary issue was that different group members would produce conflicting information, sometimes making substantial rewrites necessary. In contrast, the vast majority of our work was done together, in the same room. Group members were regularly comparing and confirming their findings with each other. Perhaps not surprisingly, we had virtually no issues with contradictory information. Though working in the same room at all times may not be necessary, it seems invaluable to regularly compare and contrast the work of different group members.

A lot of time was spent working with LSEdit. Even after spending a great deal of time trying to simplify diagrams as best as we could, trying to read these diagrams took far too long. LSEdit should include an option to abbreviate the dependency arrows, just as we did in our simplified diagram.

Finally, we had some difficulty comprehending the Firefox source code. Specifically, the comments were generally poor and uninformative, and sometimes absent altogether. It was often difficult to understand how a given file operated, and meticulous reading of the code was required. This experience reinforced the value of commenting our code properly, and taught us not to expect well-documented code in industry.

Glossary

CID Class ID.

DOM Document Object Model.

HTML HyperText Markup Language.

LSEdit Software Landscape Editor.

MIME Multipurpose Internet Mail Extensions.

UI User Interface.

W3C World Wide Web Consortium.

XML eXtensible Markup Language.

XPCOM Cross-Platform Component Object Model.

XPFE Cross-Platform Front End.

XRE XUL Runtime Environment.

XUL XML User-interface Language.

Bibliography

[1] http://research.cs.queensu.ca/home/emads/teaching/readings/emse-browserRefArch.pdf

[2] https://developer.mozilla.org/en-US/

[3] http://en.wikipedia.org/wiki/History_of_Firefox

[4] https://wiki.mozilla.org/Main_Page

[5] https://bugzilla.mozilla.org/describecomponents.cgi?product=Firefox

[6] http://en.wikipedia.org/wiki/Firefox

[7] https://developer.mozilla.org/en/XUL_Tutorial/Introduction

[8] https://developer.mozilla.org/en/XULRunner_FAQ

[9] http://www.kodewerx.org/wiki/A_Brief_Introduction_To_XULRunner:_Part_1

[10] https://developer.mozilla.org/en/XUL_Tutorial/XUL_Structure

[11] http://en.wikipedia.org/wiki/XUL

[12] https://developer.mozilla.org/en/Mozilla_Application_Framework_in_Detail#Original_Document_Information

[13] https://developer.mozilla.org/en/Mozilla_Source_Code_Directory_Structure

[14] http://www.ibm.com/developerworks/webservices/library/coxpcom2/index.html

[15] https://developer.mozilla.org/en/Using_Application_Cache