Concrete Architecture of Mozilla Firefox (version 2.0.0.3)

SEng 422 Assignment 2
Dr. Ahmed E. Hassan

July 4, 2007

Iris Lai
Jared Haines
John,Chun-Hung,Chiu
Josh Fairhead

Table of Contents

List of Figures

Abstract

In the assignment 1 report, we developed a conceptual architecture of Mozilla Firefox version 2.0.0.3 which described Firefox's fundamental subsystems. In this report, we will provide a description of the concrete architecture of Firefox's Necko subsystem, as well as a top-level concrete architecture for the entire Firefox application. We used jGrok to extract relationships from the Firefox source code, and we used the lsedit visualization tools to investigate and analyze the concrete architecture. We discovered that there were differences between our conceptual architecture and the implementation. For this reason, we have adjusted our conceptual architecture to fit our updated understanding. In addition, we will discuss a download scenario to demonstrate some of the interactions in Necko's concrete architecture.

Introduction

This report is a continuation of our study on the architecture of Mozilla Firefox. We have discussed the conceptual architecture of Firefox in the assignment 1 report, which was a high-level description of the major subsystems of Firefox as designed. This report is intended to discuss the concrete architecture of Firefox at the top level, as well as a deeper look at the Necko subsystem's concrete architecture. The concrete architecture is a high-level description of the major subsystems as implemented.

First, we studied the Firefox source code. From our investigations, we decided to modify our Firefox conceptual architecture, which will be discussed in the next section – Modified Conceptual Architecture of Mozilla Firefox. The concrete architecture of Firefox also will be discussed in this report. Second, we took a study on the concrete architecture of the Necko networking subsystem. We discovered that the implemented Necko architecture differs slightly from our conceptual architecture. The major differences were the following:

- There is no Protocol Connection subsystem in the Necko concrete architecture.
- There are extra subsystems in the Necko concrete architecture, such as Stream Converter and Necko Utility.

We will discuss a scenario of downloading to demonstrate our findings. Finally, we observed the application of several design patterns and architecture styles in the Necko subsystem, which will be discussed in the Design Patterns and Architecture Styles section.

Modified Conceptual Architecture of Mozilla Firefox

After investigating the concrete architecture of Mozilla Firefox suite, to fit our updated understanding, our group modified our top-level conceptual architecture.

In our previous conceptual architecture, our diagram did not have the display backend module because we assumed that the display backend is a part of GECKO's graphical interface. We examining the code and realizing that Display Backend is an interface that tightly coupled with the local machine. It provides widget toolkit api that can be used by the User Interface module. Also we eliminated the XPCOM module from the conceptual diagram since we thought that XPCOM is a module that should not be mentioned on the conceptual level. Conceptual diagram should focus on specifying the major subsystems. The dependency between Necko and XML Parser was also eliminated according to our concrete architecture. The modified conceptual architecture is shown in Figure 1.

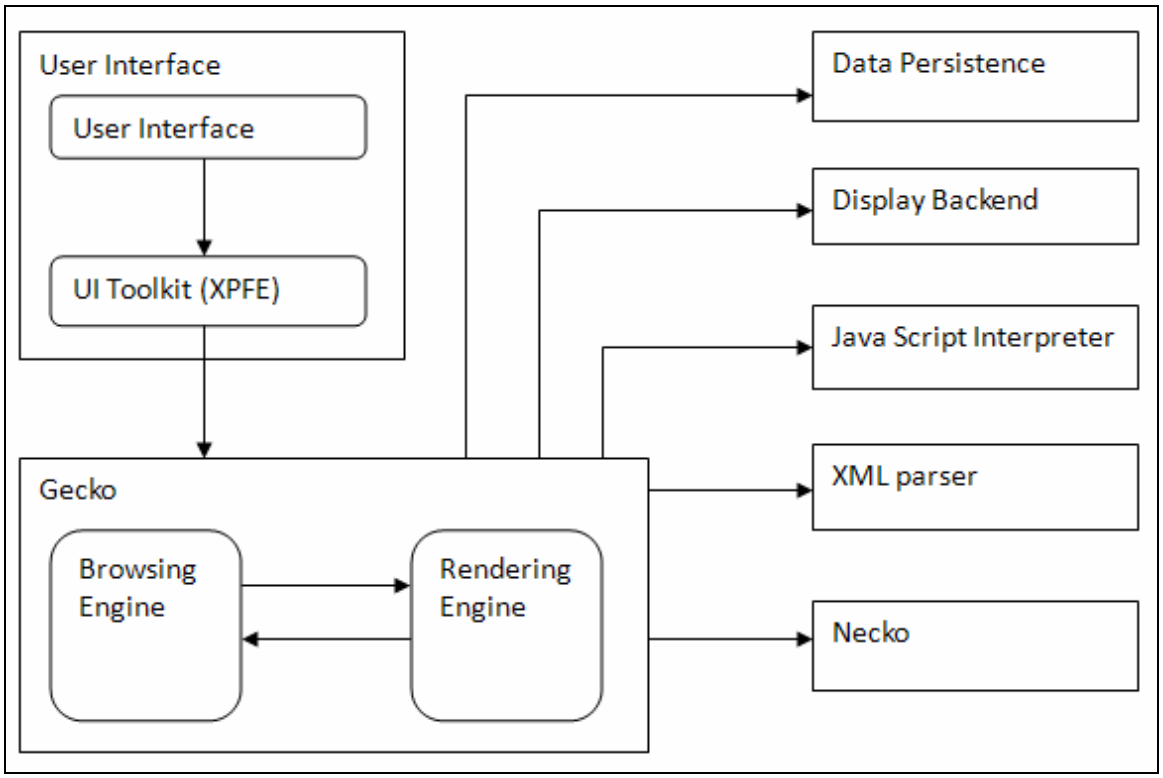


Figure 1 Modified Conceptual Architecture

Following the lab's instructions, our group used various tools such as jGrok to create containment files for Mozilla Firefox version 2.0.0.3. For visualizing the concrete architecture, we generated landscape file using these containment file and then ran the lsedit java program to show the landscape file.

The initial containment file defines the hierarchy for the source code directory structure. Since we are responsible to build a top-level view for the concrete architecture of Firefox, we edited the directory structure inside the containment file and added a few new hierarchies into it. Based on our studies and understandings, the way our group re-arranged the directory structure was to group all the relevant source folders into modules. The purpose for re-arranging the directory structure is to closely matching the concrete

architecture with our modified conceptual architecture. Figure 2 is the concrete architecture of Mozilla Firefox.

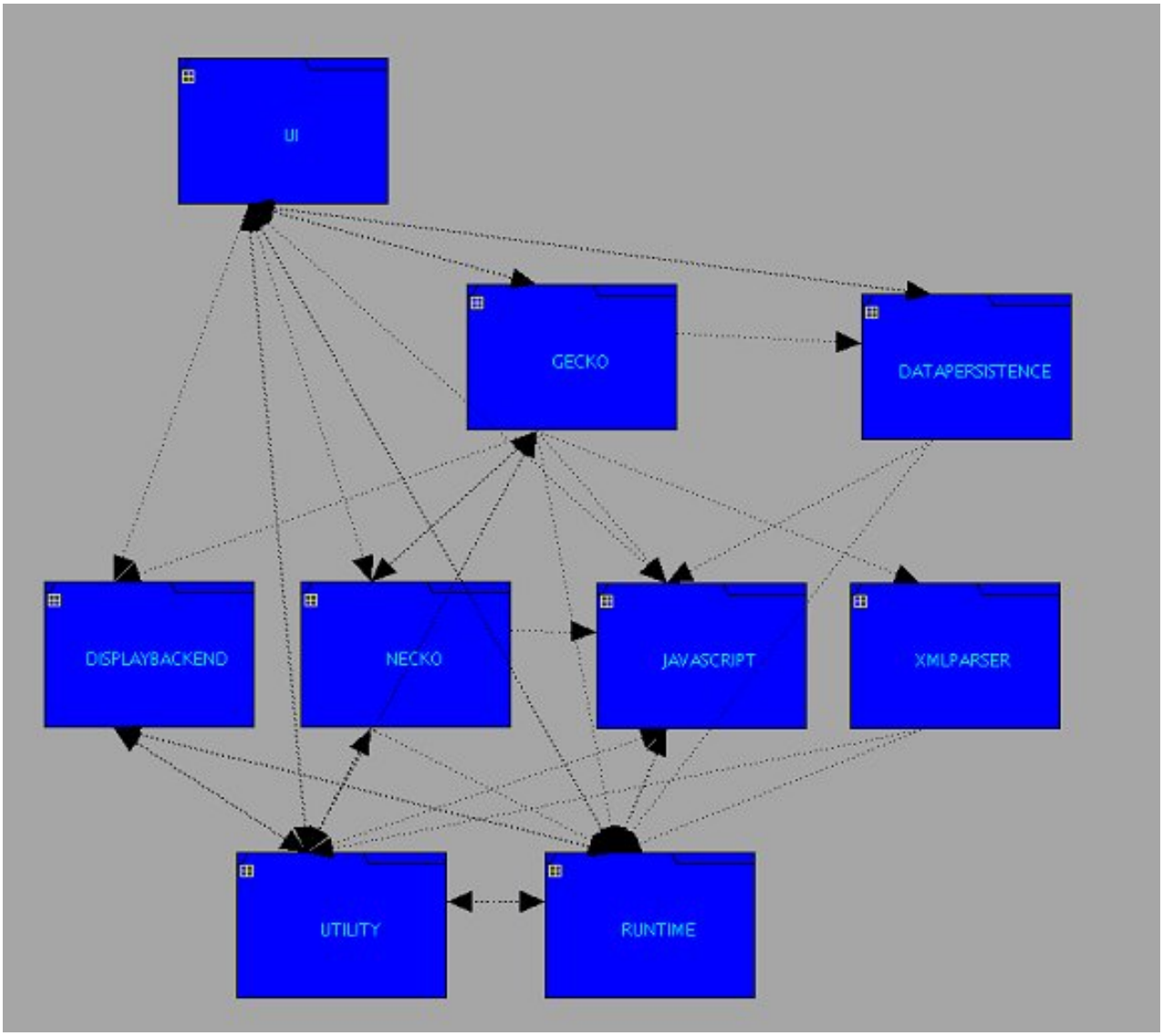


Figure 2 Firefox Concrete Architecture

Concrete Architecture of Mozilla Firefox

Based on the top-level concrete architecture in Figure 2, when our group initially opened the landscape file to visualize the concrete architecture, we noticed that it showed more relations and dependencies than the conceptual architecture. Most modules were closely related to each other. We created two more modules into the concrete architecture while

we were mapping the source code with our modified conceptual architecture. As being shown in Figure 2, these two modules are Runtime and Utility.

Major Subsystems in Firefox

Runtime

Runtime is the module that does not exist in our conceptual architecture diagram. It basically provides object framework (XPCOM) for most subsystems, and also a cross platform runtime environment (NSPR) that provides system-level services with a platform-independent API. Notice that every other modules uses Runtime module in Firefox application.

Utility

Utility is a module that provides utility functions such as encoding strings into Unicode character sets or converting data between different encoding schemes. Based on our observation, the most used function is nsUnicharUtils that provides case conversion service for Unicode characters. Most modules somehow used some functions in Utility module.

User Interface

The main difference is that the UI in the concrete architecture has forward dependencies to all modules except XMLParser, and backward dependencies from Gecko, Data Persistence, Runtime, Utility modules. However, we did not encounter anything too surprising about UI's dependencies.

Gecko

It is reasonable to see Gecko module has a forward dependency to every other module, and three backward dependencies from UI, Necko, and Utility. However, it is a surprise to see that Gecko did not have a backward dependency from Data Persistence. We did mention in our conceptual report that references to persistence storage items are accessed by the HTML code rendered through Gecko.

Display Backend

Display Backend has bi-directional dependencies with Utility and Runtime, and two backward dependencies from UI and Gecko. This observation is reasonable since Display Backend supposed to provide cross-platform interfaces for UI.

Data Persistence

Data Persistence has three forward dependencies to UI, JavaScript and Runtime modules, and has two backward dependencies from UI and Gecko.

However, as we mentioned in Gecko section, it is surprise to find out that Data Persistence has no forward dependency to the Gecko module.

XML Parser

The XML Parser module has two forward dependencies to Utility and Runtime, and only one backward dependency from Gecko. The result of this observation was expected because the XML parser is responsible for reading and processing XML documents and creating DOM hierarchies; functionality that is known to be used by Gecko. This observation also proves our original diagram was incorrect because there was no dependency between Necko and XML Parser.

JavaScript

Java Script only has one forward dependency to the Runtime module, but has backward dependencies from UI, Gecko, Data Persistence, Runtime and Utility. As far as we understand, the JavaScript engine (Spider Monkey) is an interpreter that interprets JavaScript source code and executes the script accordingly. It was surprising to see that it has so many backward dependencies from other modules. In our conceptual report, we stated that Java Script module should be strongly tied to Gecko.

Necko Concrete Architecture

Necko is a project developed by Mozilla. Necko is the networking system in Mozilla Firefox. We have discussed the conceptual architecture of Necko in the Assignment 1 report; from the previous report, we know that an URL connection request is generated and sent out from UI subsystem. And then, the request is passing through Gecko and Necko. The Necko interprets the request into a lower level commands. Necko decides what type of networking protocol will be used, how many transports are needed, and how to connect to the Internet. In the following sections, we will discuss how these are implemented in the Necko subsystem and what the Necko concrete architecture looks like.

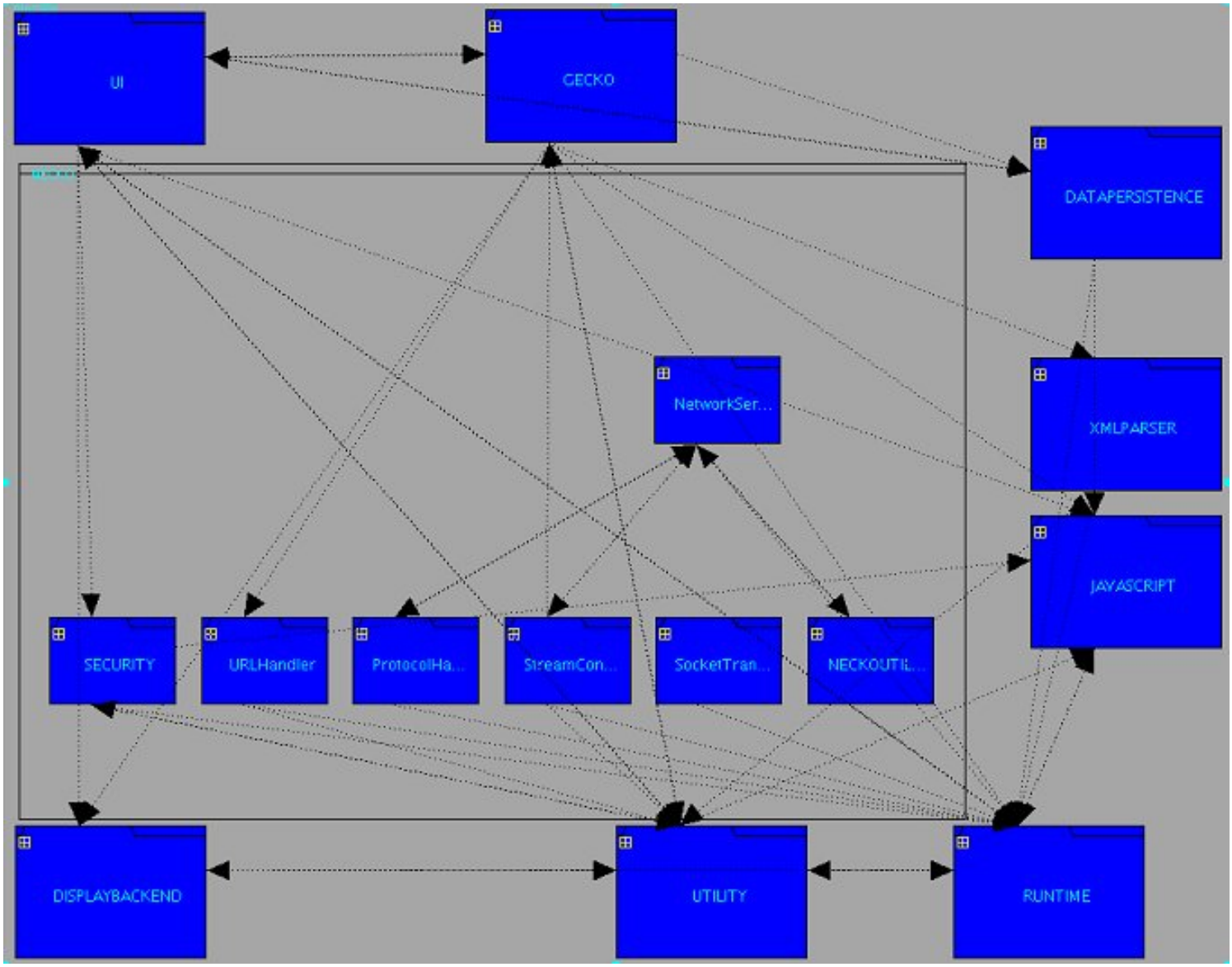


Figure 3 Necko Concrete Architecture

Figure 3 shows the Necko concrete architecture and its relationships with other subsystems. From the diagram, we can see that Necko concrete architecture includes Network Service, Protocol Handler, Socket Transport, Stream Converter, Security, URL Handler, and Necko Utility. Each module is discussed below.

Network Service

Network Service contains essential modules which are used by other Necko subsystems. For example, it contains nsIOService and nsSocketTransportService. nsIOService provides major Necko services. It manages protocol handlers and provides interface for creating URI objects from URI strings. nsSocketTransportService manages socket transport service which builds physical connections protocol handlers and the Internet.

Network Service also includes listeners, such as nsSyncStreamListener and nsAsyncStreamListener. The listeners are implemented by the consumer of an nsIChannel instance, which are implemented in Protocol Handler system. Once a listener object is created, the listener is passed to nsIChgannel.

Protocol Handler

Protocol Handler is a layer between UI and Socket Transport. The URI scheme is created in UI layer. In Protocol Handler layer, the corresponding protocol is selected according to the URI schema. For example, the Protocol Handler can handle requests that use ftp, http, or gopher protocol. The Protocol Handler layer also creates channels. Channels manage the connections to the Internet. Once a channel is created, the connection might use a socket transport or a file transport. A socket transport connects to the Internet, while the file transport connects to the file system. The Protocol Handler also manages how data is transmitted, such as in a synchronous way or an asynchronous way. In another words, how the connection threads are managed.

Socket Transport

Socket Transport is a layer between Protocol Handler and the Internet. In the Firefox implementation, there is another layer between Socket Transport layer and the Internet. That specific layer is XPCOM and NSPR which we have called the RUNTIME subsystem.

Stream Converter

Stream Converter provides stream conversion services to Protocol Handler. For example, it converts text to HTML in case the text contents are not handled by the protocol.

Security

Security uses the Mozilla Personal Security Manager (PSM) which provides a library for performing cryptographic operations. The PSM suite is built on top of NSS and consists of the PSM Daemon and the PSM Client Library. Applications use PSM functionality by calling the Client Library API. The Client Library API communicates with the PSM Daemon using the PSM Protocol over a local socket connection. The PSM protocol supports most operations that NSS support including SSL v2 & v3, TLS, S/MIME, OCSP, CRMF/CMMF and PKCS #5, #7, #11 & #12.

URL Handler

A URI is requested through mozilla/docshell component in GECKO. URI Handler in NECKO is called from the docshell. A channel from Protocol Handler is used to open the URI through the nsIExternalHelperAppService.

Necko Utility

A tool used by NECKO components that would be common to most networking library implementations, can operate independently of NECKO, and provides common services for NECKO is classified as a NECKO Utility.

DNS

DNS provides services used by NECKO through the Network Service. For example, the DNS module is used when resolving canonical names.

Cookie

The Cookie module provides services for managing cookies. A Cookie Service can be invoked which provides operations on cookies such as Create, AddCookieToList and RemoveCookieFromList.

MIME

The MIME module provides MIME type resolution services for NECKO.

Cache

The Cache module provides services for caching all documents downloaded. Every document is cached to allow back/forward, saving, view-source, and other functionality for visited pages. These services include a Cache Services which provides operations for caching such as CreateSession, OpenCacheEntry and ActivateEntry.

Interface architecture in Necko

Figure 4 is the interface architecture of networking in Necko, which demonstrates how a URL request is handled in Necko. The Application here refers to any system outside the Necko. In Firefox, this is Gecko, which talks to Necko.

In the implementation of Necko, nsIURL and nsINetService are implemented in Network Service subsystem. nsIProtocolHandlers are implemented in Protocol Handler subsystem. nsIprotocolConnection is implemented by channels. Socket transports and file transports are implemented in Network Service subsystem and Socket Transport subsystem.

Necko also handles transport threads. There is a single socket transport thread that manages a pool of file descriptors for all outstanding socket requests. This is similar to the Master-Slave design pattern. A master is monitoring all its slaves and talks to clients. In Necko, the masters are located in Network Service package, and slaves are implemented in other subsystems such as Protocol Handler.

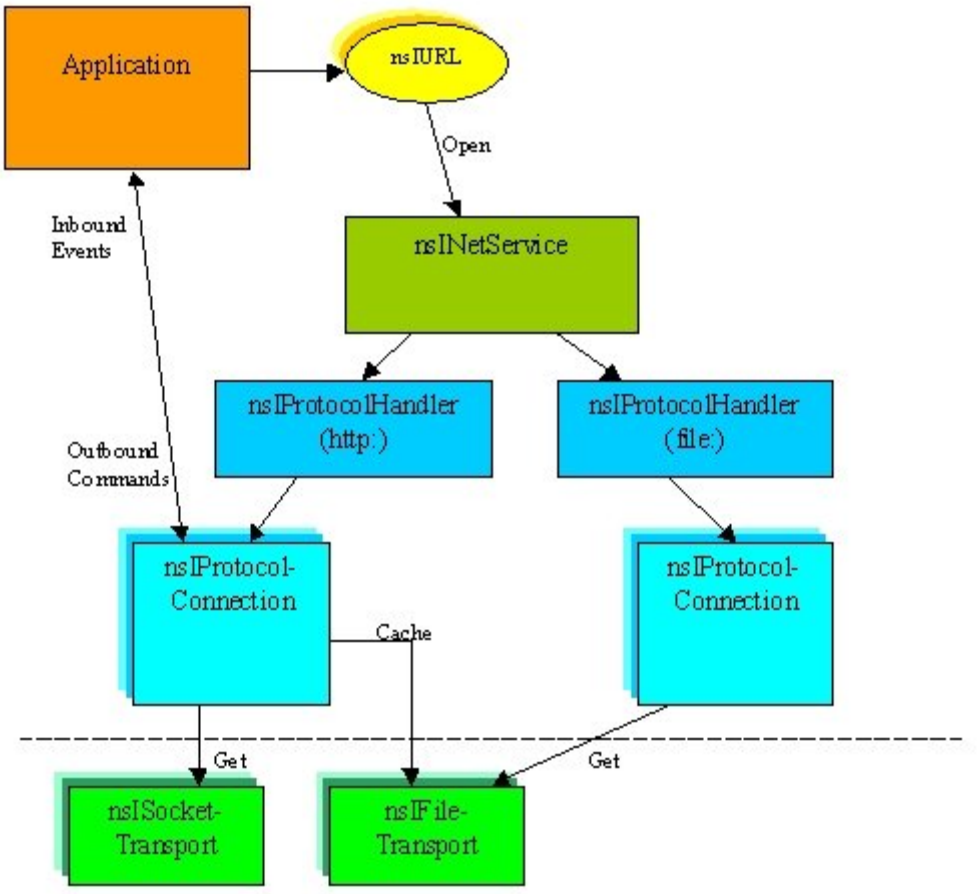


Figure 4 The interface architecture of networking in Necko

(Source: http://www.mozilla.org/docs/netlib/necko.html)

Handling downloads in Necko

Necko handles network connections, including file downloads. By using the Firefox web browser, a user can right click on a URL and choose the “Save link target as” option in the popup menu. This triggers a download in Firefox. Figure 5 shows a scenario that how this works.

In Figure 5, we can see that once the “save as” function is called, there is an object of nsIURIChecker being created. The implemented class of nsIURIChecker is located in Network Service package. The download action also calls nsIExternalHelperAppService which is in URL Handler. Once the connection is allowed to be established, the nsIDownload in Socket Transport is called. At the same time, the corresponding channel will be built. This is an asynchronous operation. While the file is downloading, the Firefox can respond to other requests.

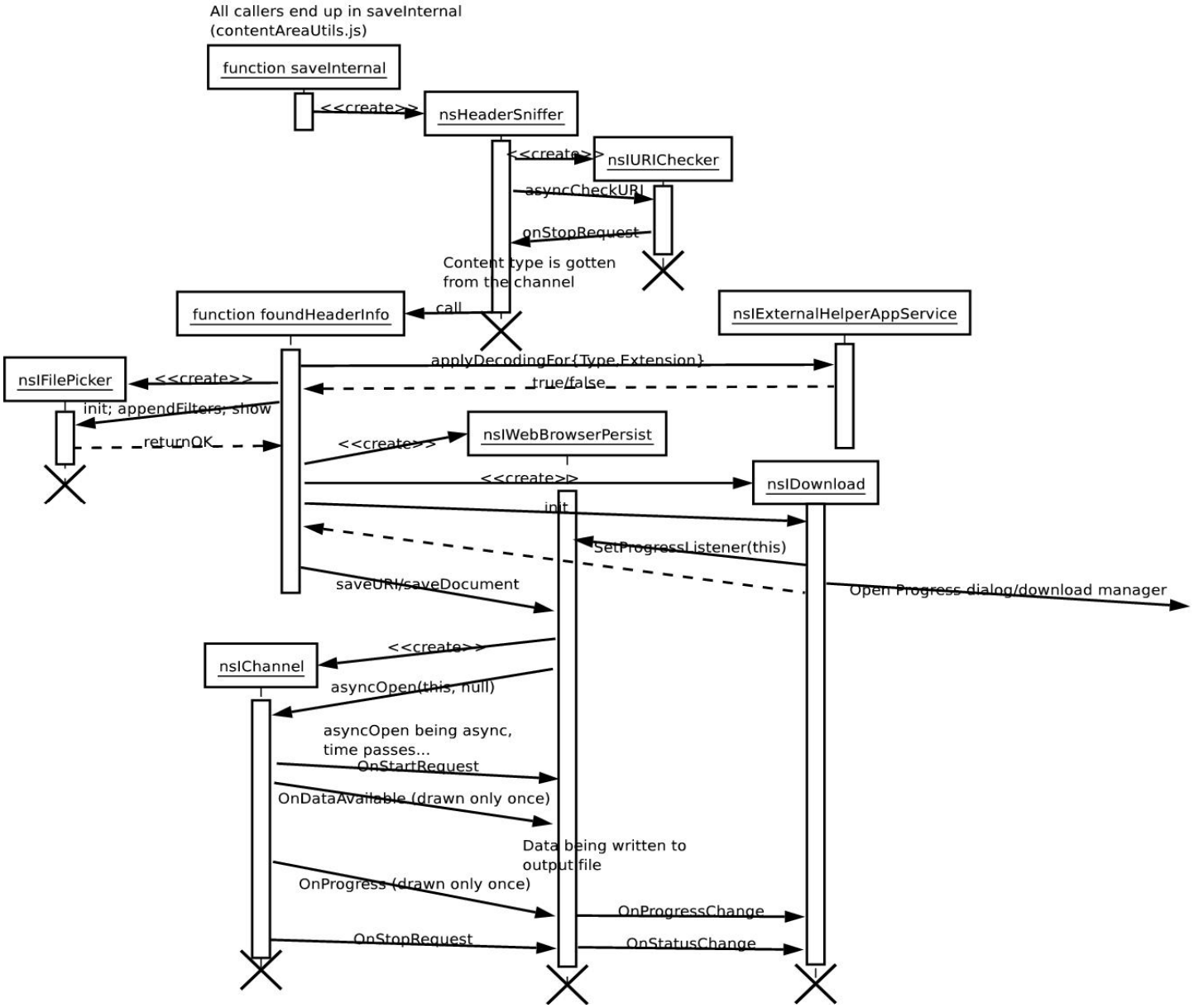


Figure 5 Downloads via "Save link target as" and friends

(Source: http://developer.mozilla.org/en/docs/Overview_of_how_downloads_work)

Necko Conceptual Architecture vs. Concrete Architecture

Conceptually, Necko contains URL, Network Service, Protocol Handler, Protocol Connection, File/Socket Transport, and NSS. In the coding level, developers were considering many situations, such as how to handle multiple threads in Necko, if there are any singletons, whether files are transported synchronously or asynchronously. As a result, the concrete architecture of Necko is different from the conceptual architecture.

The first difference is that there is no Protocol Connection subsystem in the Necko concrete architecture. As we discussed previously, the Protocol Connection is implemented by channels. There are two types of channels: synchronous and asynchronous. The channel implementations are located in Protocol Handler package. For different type of protocol, the corresponding channel is different, so that the channel can handle different protocol text contents. Second, there are extra subsystems in Necko concrete architecture, such as Stream converter and Necko Utility. Necko Utility includes DNS, Cookie, MIME, and Cache.

Design Patterns and Architecture Styles

Design Patterns

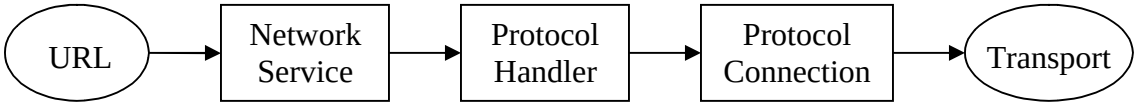
During our investigation of the Necko system's concrete architecture, the following object-oriented design patterns were encountered:

- **Observer:** Several of Necko's subsystems use the Observer pattern. Necko's cache subsystem uses an observer object to notify it when cacheable data is being received. Many of Necko's components also use a RequestObserver to notify them of the status of an asynchronous request.
- **Proxy:** The aforementioned RequestObserver object is accessed through a RequestObserverProxy.
- **Abstract Factory:** Two examples of the Abstract Factory pattern are the SocketProviderService, which is used to create instances of the various SocketProvider implementations; and the StreamConverterService, which can instantiates several different types of StreamConverters.
- **Singleton:** Two notable Singleton objects are Necko's IOService and CookieService
- **Façade:** Necko's security subsystem includes a SecretKeyFacade object, which abstracts the usage of an underlying SymmetricKey object. Additionally, many of Necko's subsystems use a Façade object to abstract multiple concrete

implementations; for example, in the protocol subsystem there are numerous handler objects for various protocols (ftp, http, gopher, et cetera), all of which implement the ProtocolHandler interface. Others such as the stream converter and socket subsystems use similar Façade objects.

Architecture Styles

The most noticeable architecture style is the Pipe and Filter style, which is employed when the system initiates a network request; the architecture provided to handle such a request looks something like this:



A URL object is passed to the network service, which determines the URL's protocol and obtains an appropriate protocol handler, which it instructs to load the URL. The protocol handler instantiates a protocol connection, which in turn produces a Transport object. This Transport object represents a physical connection to the remote data source; it will run asynchronously in a separate thread and notify the application when incoming data is received.

Necko also makes use of the Implicit Invocation architecture style for its Observer objects.

Conclusion

From extracting the conceptual architecture of Mozilla Firefox, we have learned how Firefox's subsystems are designed and organized. After taking a closer look at the concrete architecture of Firefox and Necko, we discovered differences between our conceptual architecture and the concrete architecture, and we also found out that the differences usually are caused by the way the software was implementated. By comparing our Necko conceptual architecture with the concrete architecture, we discovered that the Necko implementation has extra modules to handle situations such as what type of protocol is used, how to transmit data. We also discovered that various design patterns are used in Necko implementation, which include Observer, Proxy, Abstract Factory, Singleton, and Façade.

Glossary

Netscape Portable Runtime (NSPR)
NSPR provides platform independence for non-GUI operating system facilities, including threads, thread synchronization, normal file and network I/O, interval timing and calendar time, basic memory management (malloc and free) and shared library linking.

Multipurpose Internet Mail Extensions (**MIME**) is an Internet Standard that extends the format of e-mail to support:

- text in character sets other than US-ASCII;
- non-text attachments;
- multi-part message bodies; and
- header information in non-ASCII character sets.

Virtually all human-written Internet e-mail and a fairly large proportion of automated e-mail is transmitted via SMTP in MIME format. Internet e-mail is so closely associated with the SMTP and MIME standards that it is sometimes called SMTP/MIME e-mail.

SSL v2 and v3 Secure Sockets Layer. A protocol that allows mutual authentication between a client and server and the establishment of an authenticated and encrypted connection. SSL runs above TCP/IP and below HTTP, LDAP, IMAP, NNTP, and other high-level network protocols.

TLS Transport Layer Security. A protocol from the IETF based on SSL. It will eventually supersede SSL while remaining backward-compatible with SSL implementations.

S/MIME Secure/Multipurpose Internet Mail Extensions. A message specification (based on the popular Internet MIME standard) that provides a consistent way to send and receive signed and encrypted MIME data.

PKCS #5 Public-Key Cryptography Standard #5. The standard developed by RSA Laboratories that governs password-based cryptography, for example to encrypt private keys for storage.

PKCS #7 Public-Key Cryptography Standard #7. The standard developed by RSA Laboratories that governs the application of cryptography to data, for example digital signatures.

PKCS #11 Public-Key Cryptography Standard #11. The standard developed by RSA Laboratories that governs communication with cryptographic tokens (such as smart cards).

PKCS #12 Public-Key Cryptography Standard #12. The PKCS standard developed by RSA Laboratories that governs the format used to store or transport private keys, certificates, and other secret material.

CRMF/CMMF Certificate Management Message Formats. A PKIX format used to convey certificate requests and revocation requests from end entities to certificate authorities and to send a variety of information from certificate authorities to end entities.

OCSP Online Certificate Status Protocol. A PKIX protocol used for determining the current status of a digital certificate.

PKIX Public-Key Infrastructure (X.509). A working group of the Internet Engineering Task Force (IETF) that is developing Internet standards needed to support a PKI based on X.509 certificates.

References

[1] Mozilla: Core Modules & Libraries - http://www.mozilla.org/catalog/libraries/uriloader/
[2] Network library documentation - http://www.mozilla.org/projects/netlib/
[3] http://www.mozilla.org/docs/
[4] http://www.mozilla.org/owners.html
[5] Multithreading in Necko - http://www.mozilla.org/projects/netlib/necko_threading.html
[6] http://www.mozilla.org/docs/netlib/necko.html
[7] MDC Glossary - http://developer.mozilla.org/en/docs/Glossary#SMIME