Anatomy of a Browser

Embedded Mobile Lizards

David Llewellyn-Jones

5th November 2024

1 Title page

In this presentation I want to talk about embedded browsers. I'll come to the details of what I mean by that and why it's of interest to me, but my plan is to give you an idea about the important things that live under the skin of embedded browsers.

It's a bit of an esoteric subject, one that I thought might nevertheless be interesting. These are all things that I wish someone had explained to me before I started working with embedded browsers and while I don't expect this info will be particularly useful unless you're planning to do the same, my hope is that it may just be interesting to hear more about.

2 The Truth

Before I get started I want to emphasise that there is truth... and then there's browsers.

3 The Truth About Browsers

Everything I say today will be subjective and open to dispute. As we'll talk about I've posted quite a lot online about this topic and something that' become very clear is that there is no definitive truth when it comes to browsers. So caveat emptor.

4 Gecko

I've said I'm going to talk about embedded browsers, but it's important to understand that my personal experience is with *Gecko*, which is the rendering engine used in Firefox. Other browsers are available and many of the things I'll talk about will be transferable, but in practice I have much less experience with the others.

So Gecko. It's not new technology, having been around since 200 and the release of Netscape 6, but it remains remarkable relevant given its heritage. And there are reasons for that.

Other rendering engines you may have heard of include Blink from Chrome and Edge, WebKit from Safari and Netsurf for which the browser and rendering engine share the same name. The history of all these rendering engines is, I think, fascinating too and we'll take a look at that as well.

1. Mozilla's rendering engine used in Firefox
2. First appeared in 2000 with the release of Netscape 6
3. Alternative to Blink (Chrome, Edge, Brave, Vivaldi), WebKit (Safari, iOS, Epiphany), Netsurf (Netsurf)

5 Embedded Browsers

But first, I should explain what I mean by an *embedded* browser. You'll all be familiar with Web browsers, you almost all certainly use one on a daily basis. Possibly hourly.

But an *embedded browser* isn't a Web browser in the same sense. So what is it? Before reading the quote, does anyone want to hazard a guess?

Okay, let's read the quote and see how that compares.

The quote is from the "Web Platform for Embedded" which is an embedded browser project. There's quite a lot going on in that quote, so let me break it down a bit.

WPE: Web Platform for Embedded

What is an Embedded Browser?

`https://wpewebkit.org/about/what-is-embedded.html`

6 Embedded Browsers

So, there are several dimensions to the meaning of *embedded browser*.

1. It may be is designed to work on smaller footprint devices. For example phones, smart devices, VR headsets. That's one dimension.
2. It's likely embedded in another software application. For example, it's not uncommon for games to use embedded browsers to render textual content overlaid as part of the user interface.
3. The embedded browser itself is likely to have a *minimal* user interface. So no chrome (where chrome is a referring to the buttons and furniture around the rendered part of the browser, rather than the Chrome browser).

Am *embedded browser* might take on some of these characteristics, or all of them. In practice, many embedded browsers are built around existing rendering engines, so while they might try to cut out unnecessary functionality, the renderer itself isn't necessarily any smaller than the renderer of a standard Web browser.

I guess I should add that my experience is in relation to a browser engine that's used on a phone and which can be embedded in other applications.

7 Browser timeline

So now we understand what an embedded browser is, let's turn back to browsers more generally. I want to give a flavour of the diversity that exists in the world of browsers.

This diagram gives a visual history of just some of the browsers that have been released over the years. The actual number runs probably into hundreds, but these are some of the ones that I'm personally familiar with.

Right at the top is *WorldWideWeb*, the very first browser developed by Tim Berners Lee and his team at CERN in the early nineties. Tim Berners Lee proposed and scoped the project and did coding on the browser. He was joined by Jean-François Groff on the project who also developed the browser and Nicola Pellow who turned it into *libwww*, which is essentially the core of the engine that could be used in other browsers.

I think it's interesting to note that Tim Berners Lee is seen as the creator of the Web, but many of the crucial developers involved have been forgotten. There's info about Jean-François online, but Nicola Pellow seems to be barely mentioned on the Web outside of her Wikipedia entry.

Crucially, libwww became the first Web engine. I hesitate to say it was a rendering engine because it was more interested in handling network protocols and parsing Web content.

Line Mode Browser was the command line browser that was built from it. Amaya was an experimental browser developed at INRIA which I've kept on here as an example of where it was used. I never used Amaya myself, but I did use libwww for several projects at one point.

It was also used in NCSA Mosaic, which was probably one of the most popular browsers because it was graphical and supported images. At this point in time the standards were *de facto*, created by the browser developers to suit their specific needs.

NCSA stands for the National Center for Supercomputing Applications based at the University of Illinois Urbana-Champaign. The spun out a company, Spyglass, to try to commercialise Mosaic and one of their customers was Microsoft.

Internet Explorer was essentially built on the Mosaic codebase. Interestingly, Microsoft's licence meant they had to pay Spyglass a small recurring payment, plus royalties for every non-operating system copy of the browser sold. Microsoft went on to bundle Internet Explorer with Windows and so ended up paying Spyglass very little. Spyglass ended up suing Microsoft as a result.

The rendering engine in Internet Explorer was called Trident and as you may know Microsoft switched from Trident to Blink with the release of the Edge browser. But Edge still incorporates a "compatibility mode" which uses Trident, so the code is still in there, from 1990 up until today.

Next up we have Netscape, formed in the early nineties and which had a considerable impact on the development of HTML. Back then Netscape Navigator wasn't free or open source. But it was given away on countless CDs offered by ISPs in order to get people on to the Web. The CD would contain Netscape along with the dial-up modem configuration files for the particular ISP.

Netscape wanted to make a more standards-compliant browser, so they essentially rewrote the engine to create Gecko, released in 2000. While the Netscape code was purportedly a mess, for Gecko they made extensive use of standard coding patterns and best practice. The result was excellent code that ran like a dog on the hardware of the time. Netscape included a browser, HTML editor and Email client in one application. To make things leaner they split the app into three: Netscape, Firefox and Thunderbird.

Firefox still ran like a dog but had a cleaner user interface and was more accessible, and we still live with the echoes of that decision today. Interestingly, while the code quality was arguably a hindrance at the time, it's also arguably one of the things that allows Gecko and Firefox to remain relevant today.

I've included Servo there because I think some of us may find it interesting. The Rust programming language was developed by a Mozilla engineer and the intention is to gradually switch Firefox over to using Rust. Servo was the entirely Rust-based rendering engine developed as part of this. Mozilla folded some bits of Servo back in to Gecko, but it also lives on as a separate, independent project. It's unusual here in that it's intended *solely* as an embedded browser.

Finally there's the WebKit class of browsers. WebKit was originally a KDE project which Apple picked up for Safari because it was so lightweight (*"less than 140,000 lines of code"* as Apple said at the time). Interestingly WebKit was always built to be an embedded browser, which allowed it to be picked up quickly by other projects, including Google Chrome, Opera and Edge.

Google forked it to make Blink and this tree of WebKit-based renderers is now by far the most widely used on the Web.

1. The browser history timeline.

8 Embedding Gecko

I mentioned that WebKit was developed with embedding as first-class functionality. With Gecko, on the other hand, Mozilla has always neglected embedded browser features.

This quite is from Chris Lord. He was a Mozilla employee, although had left by the time he said this in 2016.

This is important for me because I work with Gecko as an embedded browser. As it happens, I work with a thing called XULRunner and EmbedLite, which is a development of IPCLite.

Although Mozilla completely neglects this code, it's thankfully not stripped it entirely out yet. So the embedded

capability remains there and that's what I work with.

1. It's an old quote, but holds true today.
2. Chris Lord was a Mozilla engineer at the time.
3. He makes clear he's not criticising the implementations, but rather the strategic direction of Mozilla.
4. IPCLite is another name for EmbedLite.
5. `https://www.chrislord.net/2016/02/24/the-case-for-an-embeddable-gecko/`
6. `https://www.chrislord.net/2016/03/08/state-of-embedding-in-gecko/`

9 Context

10 Working with the browser

So let me spend a little time explaining about my interest. In my previous job at Jolla developing for Sailfish OS, one of my roles was on the browser development team.

As a phone operating system Sailfish is unusually in using Gecko for its browser engine. While at Jolla I was involved with the upgrade of the browser from Gecko ESR 60 to 68, and then from 68 to 78. ESR stands for "Extended Service Release", because following every release would have been far too resource intensive.

The browser and everything that makes it up is open source. So since leaving Jolla, over the last year from August 2023 to September 2024, I've upgraded the browser again from ESR 78 to ESR 91.

This was entirely a hobby project, outside of work. I spent a few hours each day working on it. It took 339 days in total and I wrote a daily blog post about my experiences, which you can see there.

I should say that the blog is terrible literature, so definitely not recommended reading. But writing daily was incredibly helpful, both in terms of keeping a record, like lab notes which I could refer back to, and also to help structure my thoughts.

I do recommend writing a daily dev-diary, but that's a different talk.

One of the reason development took so long is that the browser has a very complex build process. It's cross-compiled in a special Sailfish OS environment that is build for x86, but which emulates aarch64 when it needs to.

That's important because the build process will often build something that's then executed as part of the build process.

As you can see a full build of the engine takes six hours 37 minutes. Even a partial build can take many hours.

So typically I'd be making a change during the day and then running the build overnight.

This is only an upgrade of course, but the challenge is that upstream, which is Firefox, will make large changes to the code, including on this occasion entirely removing the EmbedLite rendering pipeline, which is why that portion of the work took so long.

Sailfish OS applies patches to the upstream Gecko code, around 100 patches in total. So the work involves attempting to apply the patches from the previous version, fixing things up to account for changes and fixing anything that breaks.

11 Gecko dev timeline

Here's my journey over those 339 days. You can see it took 45 days before it would build and 83 days of work before I even got anything to render. By far the longest task was getting embedded rendering working, 87 days in the middle there. But I think it also gives an idea of the breadth of technologies involved in a browser.

12 Sailfish Browser

1. The Sailfish Browser isn't just gecko.
2. `sailfish-browser` provides the user interface (C++/Qt/QML).
3. `embedlite-components` provides JavaScript user interface functionality (JavaScript).
4. `qtmozembed` turns gecko into an embeddable Qt component (C++/Qt).
5. `gecko` is the EmbedLite embedded gecko renderer (JavaScript/C++/Rust).

13 Sailfish Browser

1. The Sailfish Browser isn't just gecko.
2. `sailfish-browser` provides the user interface (C++/Qt/QML).
3. `embedlite-components` provides JavaScript user interface functionality (JavaScript).
4. `qtmozembed` turns gecko into an embeddable Qt component (C++/Qt).
5. `gecko` is the EmbedLite embedded gecko renderer (JavaScript/C++/Rust).

14 Gecko Code Distribution

So that's where my interest comes from. Let's now dig in to the actual anatomy of a browser. First of all, this is what makes up a Gecko rendering engine. This is for the ESR 78 and ESR 91 codebases making up the core of Firefox.

This includes the Gecko rendering engine and the Spidermonkey JavaScript engine, plus all of the build scripts.

As you can see, perhaps unsurprisingly, most of it is C++, 13 148 748 lines to be exact.

There's also a large quantity of JavaScript: 9 116 950 lines. We'll come on to why there's so much JavaScript later.

There's actually nearly four million lines just of build scripts. Some of these are auto-generated though.

There's 3 033 345 lines of Rust. So there's still a way to go before all of the C++ has been converted to Rust.

In fact, you can see the overall code size has increased quite substantially between releases.

There's also 185 528 lines of IDL. We'll come on to what that is.

ESR 78:

1. C++: 12 795 046
2. JavaScript: 8 314 694
3. Docs: 8 134 816
4. Build: 3 691 535
5. Rust: 2 652 738
6. IDL: 183 404

ESR 91:

1. C++: 13 179 126

2. JavaScript: 9 130 130
3. Docs: 8 336 234
4. Rust: 3 033 345
5. Build: 3 497 457
6. IDL: 185 528

15 Chromium Code Distribution

These are the numbers for Chromium for comparison, so this is the Blink rendering engine and the V8 JavaScript engine.

As you can see, it's overwhelmingly C++.

There's also some Java and Objective-C. Those are for the chrome, the user-interface elements of the browser and are likely targeted at different operating systems.

1. C++: 79 982 423
2. Docs: 34 517 513
3. Build: 15 319 659
4. JavaScript: 8 564 048
5. TypeScript: 3 059 540
6. Rust: 3 005 379
7. Go: 2 628 291
8. Java: 2 312 098
9. Config: 1 918 957
10. Obj-C: 1 424 269
11. Other code: 1 360 893
12. IDL: 449 886
13. WASM: 438 718

16 Gecko Patches

This slide is a bit of an indulgence. It shows the changes, in lines of code, needed to get Gecko working on Sailfish OS. Note that unlike the other graphs this is a logarithmic scale.

So, just considering the ESR 91 changes, there were 22 476 C++ lines added and 631 removed.

For JavaScript there were 170 lines added and 43 removed.

For Rust there were 498 lines removed and 180 lines added.

So these are quite substantial changes to the codebase, but the large majority of the code is left untouched.

ESR 78 Language: Added, Removed

1. C++: 22 726, 606
2. Docs: 510, 2

3. Build: 28 558, 20 350
4. JavaScript: 158, 6
5. Rust: 544, 175
6. IDL: 29, 3

ESR 91 Language: Added, Removed

1. C++: 22 476, 631
2. Docs: 508, 12
3. Build: 19 320 11 090
4. JavaScript: 170, 43
5. Rust: 498, 180
6. IDL: 39, 6

17 Basic Browser Structure

Let's turn to what all this code is for.

I've pulled out the six main components of a browser as I see them.

At one end we have the protocol client. This deals with the networking stuff, opening sockets and making HTTP and HTTPS requests. It might also handle file requests, FTP requests and others.

Then there's the JavaScript engine. The Firefox engine is Spidermonkey. The Chrome engine is V8.

Quite apart from HTML browsers have to handle a whole host of other media types: videos, images, audio. So we also need a whole media decoder stack as well. In practice this is provided by libraries and operating system functionality. For example, Android has its own Droidmedia stack, libpng will tackle PNG images and so on. But they all need to be integrated.

Then we get on to Web pages and HTML where we find the DOM, or *Document Object Model*. This is a tree of data structures within browser memory that represents the structure of the HTML document. This structure is standardised by the W3C, but different browsers implement the structure in different ways of course. The DOM is a really critical part of the browser.

You'll recall we talked about *libwww* earlier as the original browser engine. It handled layers 1, 2 and 4, although at that point in time the DOM wasn't standardised.

Next we have the actual layout and rendering engine. You may know of the *box model* which arranges all of the elements on the page in terms of the space they take up as rectangles with margins and padding. This is handled here, along with the actual rendering, for example using OpenGL, Vulcan or something else depending on what a platform supports.

Finally we have the chrome, which are the user interface elements that are separate from the Web page itself, like the forward and backward buttons.

18 Browser components

Here we see how these different elements interact.

At one end the Web server interacts with the Internet.

At the other end the Chrome interacts with a human via the operating system's compositor.

The Renderer takes the DOM and renders it to a suitable backend that the compositor can support.

The JavaScript engine works away in the background, crucially updating the DOM in order to have an effect on what gets rendered.

On Gecko we also have these things called `nsDocShell` and `nsWebBrowser` which are essentially the interfaces between the backend and the frontend code.

19 Embedded Browser Surface

So those are the internals of the browser.

But when talking about an embedded browser, as a developer we want to embed the browser into some other application code.

So we don't want to have to deal with all the browser internals. In fact, we want to know as little about it as possible.

What we really care about is the surface that the browser exposes. The API that we interact with in order to get things to work.

Here I've listed the things that, in my understanding, makes up the surface of the browser.

There's the rendering canvas, which is the bit that ends up on screen showing a Web page.

There has to be some kind of event loop from the application that the browser has to hook in to. If we're using an embedded browser as part of a toolkit like GTK or Qt, we may not have to worry about that, which is great.

Then there are browser actions, things like providing a URL for the page to show, the API equivalent of a back and forwards button and so on.

We want to be able to give the browser JavaScript to execute and this may take one of two forms.

You'll recall that I mentioned large parts of the browser internals are written in JavaScript. This is called *privileged* JavaScript because it has access to all the internal functionality of the browser. If we're embedding a browser into another app we might want to extend the capabilities of the browser by adding our own privileged JavaScript.

But we also may want to interact with the Document Object Model of the Website being rendered. So we may also want to be able to add our own JavaScript that executes in the DOM. For security reasons this has far fewer privileges. In fact, it essentially has the same privileges as the JavaScript of the site itself.

The difference is that the browser also needs to support some for of message passing between the privileged and in-DOM JavaScript, or indeed to other parts of the browser code.

If our embeddable browser supports these features, then we're basically laughing. With all this, we can do an amazing amount of stuff.

In particular, we can basically make the browser seem like a fully integrated part of the application.

20 Embedding APIs

Now the interfaces to different embedded browsers are all different in the details, but in practice they all offer this quite similar set of functionality.

Because of the C++ underpinnings, these browsers all offer object-oriented APIs which tend to provide some common classes.

There some kind of class that acts as a conduit for configuration and settings.

There are view widgets, allowing an application to embed multiple browsers all with the same engine. Conceptually this is similar to tabs.

These view widgets will offer the simple and most basic Web controls, like setting the URL, going forwards and backwards.

They all offer an API for executing JavaScript. It's worth noting that there are typically several different

interfaces for this, depending for example on whether the JavaScript should be privileged or executed in the DOM.

Finally there are some quite complex message passing interfaces to allow the front-end to communicate with the different JavaScript contexts.

For example, Gecko supports messages, notifications and events, all of which offer means to pass information from one part of the system to another.

We'll briefly look at the actual APIs for three different embedded frameworks: CEF, WebEngine and WebView.

21 CEF API

First up is CEF, Chromium Embedding Framework. Although Blink is designed to be embeddable, in practice actually using it as an embedded rendering engine, alongside the V8 JavaScript engine, is pretty horrific. CEF provides a much more accessible way of doing this.

CEF is different from the other examples we'll see because the others – WebEngine and WebView – are integrated into widget frameworks. CEF on the other hand is designed to work with many different frameworks.

Because of this it provides a core API, but then it also offers example code showing how to embed the renderer it a mixture of different frameworks.

As a developer, if you want to use it, you're expected to take that existing code and either integrate it into your application, or build on top of it. So you end up essentially using their bootstrapping code to allow Blink to be embedded into your application.

It's quite cumbersome, but nevertheless is widely used. For example, Steam, Spotify and MATLAB all use it, but there are many more. As far as I'm aware Google isn't involved in its development.

The interfaces we see here are:

1. `CefBrowserHost`: The core browser class that offers non-page specific functionality and the creation of `CefBrowser` objects.
2. `CefBrowser`: A wrapper class used to control a particular view.
3. `CefFrame`: Allows deeper interaction with a Web page. You'll notice this is the interface used for executing |JavaScript and sending messages to the browser window.
4. CefV8Value: An interface for accessing JavaScript objects from C++. This can hold many different data types, including Ints, Floats and Strings. And also functions.

22 QtWebEngine API

We can compare this to the Qt WebEngine API. This is part of the Qt Widget framework and so the WebEngine is far better integrated as a system widget. It offers a powerful API, but one that's much reduced compared to CEF.

The Qt WebEngine provides two main classes.

1. `QWebEnginePage`: Provides access to core functionalities including execution of JavaScript. You can send JavaScript to be executed which can interact with the DOM and return a result.
2. `QWebEngineView`: Provides the view onto a Web page, including setting the URL, navigating, etc. As you can see, the `QWebEnginePage` is accessed via the `QWebEngineView`.

23 Sailfish WebView API

Finally, this is the Sailfish WebView API. Here we have three key classes.

1. `WebEngine`: Provides access to the core functionality of the renderer, including sending and receiving notifications from the privileged code and injecting privileged code into the system.
2. `WebEngineSettings`: Allows global renderer settings to be controlled.
3. `QuickMozView` which is sub-classed to a class called `WebView`: represents a single widget in an application for a page. You can see it's also possible to execute non-privileged JavaScript in the DOM context and to navigate and so on.

24 Embedded Browser Examples

Those are the basic fundamental APIs provided by the three frameworks. To me they look quite similar. Structured slightly differently but offering very similar functionality. And of course that shouldn't be surprising.

Here are some examples of them in use; the code for these is all in the presentation repository which I'll share at the end.

These are really simple applications, but they demonstrate the key functionalities:

1. Integration with the native user interface. All of the chrome here is provided by native graphical user interface elements; GTK widgets for CEF, Qt widgets for Qt WebEngine and the Sailfish WebView.
2. Pressing the button runs a simple piece of JavaScript code that's been injected into the page. This walks the DOM, collects some basic statistics about it, such as the depth and width of the tree, and gives every element a red border as it goes. It's the same JavaScript in all three applications. So this demonstrates how you can inject into and execute JavaScript inside the engine from the native GUI.
3. The widgets along the bottom show the returned information, sent through the messaging system. This shows how the JavaScript running in the engine can interact with the native GUI.

Each of these applications contains a bunch of boilerplate code that any application would need. For the Qt WebEngine and WebView applications, integrating the embedded browser took a minimum of five lines of code. Developing the CEF app was a lot more challenging. I started with their GTK example, which is already 137 source files and had to make significant changes to it.

25 JavaScript DOM walker

I don't want dwell on this, but this is the DOM walking code that was injected into each of these applications.

It's super-simple, it walks the tree recursively, collects a bunch of data, then returns the results in this `global_ctx` structure.

I've put it here in case anyone wants to refer back to it.

26 Interface Definition Language

We've seen the sort of API that the various embedded browsers offer, we've seen how to use them in practice. Now I want to go back deeper into how the browser works internally, because I think this really helps contextualise how to develop for an embedded browser.

There are two things I want to talk about specifically. The first is the Interface Definition Language that's used to allow interaction between all of the different supported languages and the second is inter thread communication that's used to deal with concurrency and sandboxing.

Both Gecko and Chromium have a similar Interface Definition Language. This is a language that's used to allow interaction between native code such as compiled C++ or Rust, and the interpreted JavaScript.

It allows the browser developer to specify classes and functions – interfaces – which can be executed from JavaScript or from C++, irrespective of the language they're written in.

I'll show you some examples, but the idea is that you write your interfaces using IDL. You can then pass this through a tool to generate C++ header files or Rust traits which you can then implement.

JavaScript and C++ have very different basic types. For example JavaScript doesn't distinguish between integers and floats. So there has to be a type system in the IDL that maps onto both, which is what we see in this table.

27 Interface Definition Language

Here's an example of an IDL file. This is an interface for triggering browser prompts like password boxes to appear on screen.

At the top we see the IDL interface which defines one method `getPrompt()` which takes in some parameters and returns a result.

The actual interface is far more complex, but I just picked out one part of it.

At the bottom we see the C++ generated for this.

The `nsISupports` inheritance means that this class supports some basic introspection capabilities.

We can see that it's set to allow use from JavaScript code.

Then we have a C++ signature for the method, which we'd then have to go away and implement.

28 IDL implementation

Here's an implementation of this interface as it might look in JavaScript.

Again, there's an implementation of the `getPrompt()` method which in this case has an implementation that returns a prompt. This is code actually taken from Gecko.

On the right we have an equivalent implementation in C++. This is code I fabricated, it's not actually from Gecko since you'd never need two implementations of the same interface.

In both cases the `observe` interface is for message passing, which we'll come to.

29 Invoking IDL code

Finally here we see an example of how we'd invoke the methods.

At the top we have a JavaScript version. First we get the service that implements the interface, then we invoke the method on the returned object.

At the bottom the C++ equivalent. We get the service using the introspection offered by `nsISupports` and then execute the method on the returned object.

Being able to invoke C++ and Rust code from JavaScript, and JavaScript code from C++ and Rust, is a critical architectural feature of the browser. Without this, writing a browser would be immeasurably more challenging.

Let's take a look at this in action.

You may be familiar with the developer console on Firefox. This provides access to in-DOM JavaScript execution.

But there's also a secret console for *privileged* JavaScript execution.

You open it with the `Ctrl-Shift-J` shortcut.

Then we can access these interfaces we were just talking about.

```
Ci = Components.interfaces
```

```
Services.prompt.QueryInterface(Ci.nsIPromptFactory)
result = Object
Services.prompt.getPrompt(self, Ci.nsIPrompt).promptPassword(null, "Please enter your Primary Password.", 
console.log(result.value)
```

30 JavaScript Basic Types

That gives a flavour of how the IDL works.

A quick diversion now into the JavaScript type system.

I mentioned earlier that JavaScript doesn't distinguish between floats and integers. But in fact that's not quite true. Under the hood JavaScript stores numbers as integers if they're whole values and only converts them to floats if it needs to.

This here is code from the Spidermonkey Just In Time compiler and I think it's really fascinating.

All base JavaScript types are stored in a single 64-bit word. That includes:

1. 64-bit floating point values.
2. 32-bit integers.
3. Memory addresses.

But doubles are 64-bit. Pointers are 64-bit. How does it store all of these different things in a single 64-bit word?

The answer is that 64-bit floating values, as defined in the IEEE 754 specification, support massive redundancy in the NaN value. That's the number used to represent things like zero divided by zero.

There are in fact $2^{52} - 2$ different floating point values, all of which constitute a NaN. There are reasons for this, not least as a means of simplifying implementations in hardware.

But it means that Gecko has space to store a bunch of flags for different data types inside that redundant space. Alongside these flags there's then space for all of the 32-bit integers and also 47-bit pointers. The browser then uses mmap to ensure that pointers never needed to be larger than 47 bits.

So it turns out it's possible to store all of the basic JavaScript types, including all of the 64-bit floats, in a single 64-bit word.

31 Inter-Thread/Process Messages

Alright, that's the Interface Definition Language.

I also want to talk about Gecko's Inter-Thread or Inter-Process Messaging framework.

Browsers are massively multi-threaded. As you may know, Chrome took this a step further by placing each tab in a separate process.

Consequently the browser needs to be able to communicate between threads without concurrency issues.

It uses IPDL for this.

IPDL is inspired by the *actor model*. In this model an actor can spawn other actors which it can communicate with asynchronously.

In the theoretical model the only communication allowed is with spawned processes.

In Gecko, there are defined parents and children and these are virtual interfaces or traits, so there's no requirement for them to spawn one another.

Actors bind to an endpoint which is bound to a thread. A parent can then schedule a task to run on a different thread and *vice versa*. It's like RPC, but with the network replaced by the inter-process communication.

Let me just add that there are many variants of actor models and the version in Gecko appears to align closest with what's known as the *Fog Cutter* actor model.

1. IPDL quick reference: `https://firefox-source-docs.mozilla.org/ipc/ipdl.html#ipdl-syntax-quick-reference`
2. Similar to a "Fog Cutter" actor, see Carl Hewitt, "Actor Model of Computation: Scalable Robust Information Systems," 2015, `https://arxiv.org/abs/1008.1459`.

32 Inter-Thread/Process Messages

Here we see some of the key characteristics, some of which are qualities you want in a messaging system and some of which are designed to avoid deadlocks.

So message ordering is preserved.

Messages can be sent from parents to children or children to parents asynchronously, but only child actors can send synchronous messages to parents. The idea here is to avoid deadlocks, because if both could send synchronous messages you could end up both waiting on the other indefinitely.

This doesn't entirely prevent deadlocks because anything can be defined as a parent and anything as a child, simply by implementing the correct interface, so you can still have two threads waiting on synchronous calls between one another.

This isn't possible with the theoretical design of the actor model.

From the perspective of the developer they all just look like asynchronous method calls which is convenient.

So what you do is write an IPDL file, send it through a tool which generates C++ interfaces for the parent and child which you, as a developer, then implement.

Let's see some quick examples.

1. IPDL quick reference: `https://firefox-source-docs.mozilla.org/ipc/ipdl.html#ipdl-syntax-quick-reference`

33 Inter Process/Thread Definition Language

Here's an interface as defined in IPDL.

The `child` methods are those that can be called from the parent and execute on the child's thread.

Contrariwise the `parent` methods are those that can be called from the child and are executed on the parent's thread.

Passing this through the tool generates these two interfaces which require implementation.

34 IPDL Implementation

As a developer we then have to subclass these interfaces and implement them, just like the code on the left.

Under the hood, we end up with generated code that looks like this.

This code here runs in the parent's message loop on the parent's thread. When it receives a message it figures out what method is supposed to get called using a big `switch` statement. It deserialises the parameters and calls the function with them.

This is a really common pattern in the Gecko code. You see it everywhere.

35 Working with Gecko

Alright, that wraps up my talk. I hope I've given a flavour of what working with embedded browsers – and Gecko in particular – is like.

The wonderful thing about Gecko, apart from the fact it's open source and a very mature codebase, is that it provides a an entire ecosystem of ideas.

There are very few software applications that offer such a broad set of functionalities, outside of operating systems.

Plus browsers are constantly developing, introducing new ideas and building on new technologies.

1. Mozilla Gecko logo
2. Mozilla Public License Version 1.1
3. `https://commons.wikimedia.org/w/index.php?title=File:Mozillagecko-logo.svg&oldid=683398296`

36 Picture

37 Further info

Here are all of the main links associated with this talk in case you want to know more. The slide source repository contains the slide source and example code and I'll post a link to it on the tech talk archive page.