International Journal of Computer Applications (0975 – 8887)
Volume 174 – No. 29, April 2021

A Proposed Design for Cross Platform Applications using Web Technologies

Jebreel Alamari
Department of Computer Science University of Colorado at Colorado Springs
Colorado Springs, United States

C. Edward Chow
Department of Computer Science University of Colorado at Colorado Springs
Colorado Springs, United States

ABSTRACT
Web applications are easier to build, maintain, and distribute than native applications. With new developments in JavaScript engines and the introduction of WebAssembly, web applications can execute CPU-intensive tasks at 0.9 the speed of C/C++. Dozens of natively implemented APIs such as WebGL and Audio API are now exposed to JavaScript through the browser. With the help of these APIs, we see more and more web-based applications and games. This paper presents a novel design to get the most out web technologies. It aims to bring freedom of the web to desktop applications. Developers will never need to use proprietary programming languages or frameworks to implement their ideas. Even with existing cross platform frameworks there is still a learning curve that developers need to go through. However, in this proposed design the development is done entirely using standardized web technologies.

General Terms
Cross-platform applications, Computation in the browser, WebAssembly performance, Lighttpd webserver

Keywords
web applications, web browsers, JavaScript, WebAssembly

1. INTRODUCTION
According to World Wide Web the goal for its development is to give a universal access to a large universe of documents. It is the most powerful global information system that is accessible to everyone with an internet connection [1]. The WWW (World Wide Web) is one of the most important human inventions of all time. It connects the world in a way that was not possible before. Blogs, social networking websites, and forums are just a few examples of what is possible through this incredible system.

Web browsers are the window to this rich world of information. Because every computer needs to access the web, every single one of them comes with a preinstalled web browser. That makes web browsers the most ubiquitous software in the history of computing.

The web is rapidly changing and so is the browser. In addition to fetching and rendering HTML documents, the browser manages databases, performs cryptographic operations, and runs CPU-intensive tasks.

In this research project we are trying to show that the browser can change the way we build software. It is the solution for making cross platform applications that do not require learning a special programming language or complex GUI (Graphical User Interface) libraries. The only tools a developer needs are HTML, JS, CSS, and WebAssembly.

We argue that existing cross platform frameworks such as NW.js [2] and ElectronJS [3], are not the solution to this problem. Using such frameworks adds an extra load for the development team. Sometimes, they may end up writing native code to speed up the performance of their applications [4].

In some situations, it could increase the budget for making apps, if we consider training expenses to use one of these frameworks.

2. RESEARCH MOTIVATION
First, we believe if such a design exists, it will make building applications simple and less expensive. According to Kinvey report in 2017, the average cost of building an enterprise level application is about \$270K [5]. The driving cost of building applications could be summarized in two points.

2.1 Building and maintaining different code bases
An enterprise level application must be available on all major platforms, which means developing and maintaining different code bases in different programming languages. Web applications on the other hand are written once and run everywhere. So, we cut the cost of building different versions and maintaining different codes. If the browser supports a feature, it is available in all devices regardless of the operating system.

2.2 Applications Distribution
One of the most popular ways for distributing software is through the app store. Unfortunately, the app store itself is owned by a company that is considered a competitor. Competing against the owner of an app store is not a winning battle. In addition to sharing the profit of the application with an app store maintainers and paying annual fees, they may for some reason remove applications from the store. Also, they can develop similar applications and tweak their search algorithms to show their applications in the search results [6].

Applications in this proposed design are available on source code sharing sites such as GitHub and Sourceforge. Applications can also be hosted on the IPFS (Inter Planetary File System) network which makes them available everywhere and anytime without anyone's control [7].

As the code of web applications can be viewed in the browser console, it is easier to inspect what the code is doing other than serving its primary purpose. Even if the application was downloaded from an unknown source, it is considered less malicious compared to binaries. That makes distributing applications is as simple as sharing a hyperlink. Secondly, we

think it is the time to build fast applications with high-performance web technologies such as ASM.JS SIMD.JS, and WebAssembly.

According to multiple studies of the performance in the browser [8][9] , the performance of JavaScript is getting close to the performance of native code. ASM.JS, a JavaScript subset, is even more efficient with handling CPU-intensive operations. WebAssembly is the ultimate choice for running computation on the browser at the speed of C code in most cases. Even though WebAssembly can run outside of the web [10], but to run it in the browser, we need to serve it from a webserver. Therefore, in this design, we introduce another effective use case for WebAssembly.

Third, adapting web applications may Improve the security of desktop applications. The web browser is unique software that was built to fetch foreign code and execute it securely without asking questions. The browser sandboxes foreign codes to minimize the damage that could happen to the host machine if that code were malicious. In our design, we benefit from this feature to enforce isolation between processes in the memory [11]. We do not have to reinvent the wheel because the browser has been doing that for years. Here are some security features that can be added by adopting our design.

2.3 All web application components run on the same device, there is no back-end

A great example of SaaS (software as a service) is Google Docs and Office 365. These applications are powerful and convenient. The cloud backend of these applications is the source of power but also the source of security issues, for two reasons:

2.3.1 Service providers always have access to clients' documents.

There is no guarantee that these service providers are not using users' data for unknown purposes [12].

2.3.2 Files are transferred through the network.

that means files can be intercepted during the transfer time. Even though the connection is encrypted but just having the files on the network is less safe.

In this design, even though we are using web applications, the frontend and the backend are both on the same machine. Therefore, data never leaves the computer.

2.3.3 In this design we can limit data collection.

Native applications can use communication capabilities on users' machines if they are granted required permissions to do that. These permissions are sometimes granted by default. They can also access different services such as GPS. If an app has access to these services and has access to the Internet, there is no reason to assume that the app is not sending data to some remote server.

Unfortunately, Data collection is not only performed by the Operating System vendor but also by third party applications [13].

In this design we again use a browser feature for good. All major web browsers enforce SOP (Same Origin Policy). It states that a web application cannot communicate with another host other than the origin of the application [14]. In this design the origin is the localhost. Even with the device connected to the internet, the application will never be able to communicate with an external server.

The browser also prompts the user to allow access GPS, camera, or microphone whenever the application needs them. There is no permanent access to local devices or sensors.

With android 11 these permissions are revoked after a certain amount of time, but this feature has been available in the browsers for years [15].

2.3.4 Better Memory management with sandboxing.

Because browsers run code from external origins, they enforce sandboxing policy [11]. Foreign code is not allowed to access memory space that belongs to other processes.

Finally, web applications are portable. In other words, whenever there is a browser installed on the system, the application can run.

3. RELATED WORK

3.1 NW.js

It is considered a cross platform solution to make native apps using HTML5, JavaScript, CSS. It is based on Chromium project and Nodjs. It uses WebKit rendering engine to render html elements. It has a complete support for Node.js APIs and third-party modules. It allows calling Node.js modules directly from the DOM (Document Object Model) and Web Workers [2].

The main drawback of this framework is the size of the application. About 70-80 Megabyte of data is necessary for any application developed using this framework. This data is the NW.js itself.

3.2 Electron JS

It is similar framework to build apps using HTML5, JavaScript, and CSS. It was used to build popular desktop applications such as Twitch, WhatsApp, and Visual Studio Code [3].

This framework suffers from the same problem NW.js has which is code size.

Both frameworks claim to use web technologies only to make desktop applications. However, knowing HTML, JavaScript, and CSS is not enough to start making desktop applications with these frameworks. Developers need to spend hours to navigate through the documentation just get started.

Learning one of these frameworks may be equivalent to learning a new programming language. From business perspective this is an expensive process.

In our design, there is no complicated development setup, no special IDEs, and no dependencies. A simple text editor and some knowledge in HTML and JavaScript is enough to build a desktop application.

Both frameworks use Nodejs to interact with the filesystem. That means, whenever a single application is running, there are two instances of V8, JavaScript engine, running on the same machine. One running inside Nodejs and the other inside

the browser. It becomes worse if we have multiple applications running at the same time.

In our design we have only one instance of the webserver that serves all applications installed on the machine.

In addition, in our design, we kept the structure of traditional web applications. Backend scripting languages such as php or python can be used.

Both frameworks use chromium to render the GUI and execute code. In our design, there is no restriction in which browser to use.

4. PAPER CONTRIBUTION

In this paper we looked at the problem of making universal application using web technologies from a different perspective. We came up with a simple design that uses existing standardized technologies that are not under anyone's control. We do not intervene in how developers should organize or develop their code, nor do we intervene in how users use installed applications on their machines.

We implemented a prototype of this design that allows users to install/uninstall, view, and launch web applications as if they are native applications.

5. DESIGN

In figure 1 we can see the overall view of a typical web application. The main idea for our design is simple, which is to bring the backend and the front-end of the web applications to the same machine as shown in figure 2.

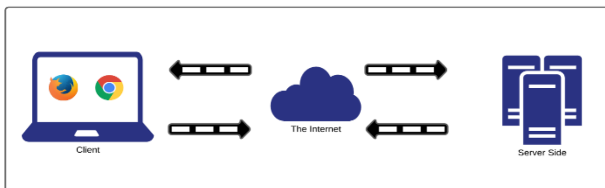

Fig 1: Web Applications Architecture

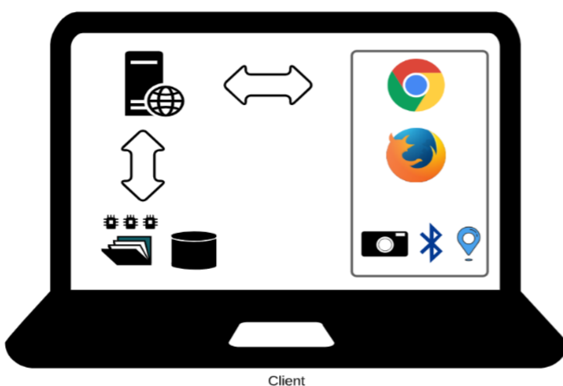

Fig 2: Web Application Local Installation

The browser in this design will take care of rendering user interface and performing computation. The web server will take care of handling file system and database access. The browser does not have direct access to the local file system, that is the responsibility of the web server. Doing this will end the need for file system API (Application Programming Interface) in the browser.

As shown in Figure 1,2 there are multiple components that an application needs to run. Making these components available for an app is the challenging part of his design. Typical computer users are used to installing apps that work on the desktop without any extra settings or configurations. Therefore, in this design we implemented an application manager that takes care of installing, uninstalling, viewing, and launching apps.

The application manager is implemented in python using Tkinter GUI library. We chose Python for a couple of reasons. First, application manager tasks are simple, so performance is not an issue here. Second, as python is a scripting language, the source code of this application manager can be viewed and inspected by users for more transparency about the inner workings of the code. In our future work, we plan to replace Tkinter with PyQt for a cleaner GUI.

5.1 Application Manager Tasks in detail

5.1.1 Installation

Any web application can be installed without any restrictions. However, the developer needs to have all the code in one folder and compress it. The application manager expects a compressed file with .zip format. The rationale behind using zipped files is that GitHub repositories can be downloaded as zipped files. Therefore, we thought about making installing codes from GitHub more convenient. In addition, we needed all the files to be bundled together in a single file.

After having the web application in the local machine, user can click in "add application button" and navigate through the file system and pick the zipped file. Then the application manager will perform the following steps in order to install the application successfully on the local machine.

5.1.1.1 Decompress the file

Because the installer expects a compressed version of the application folder.

5.1.1.2 Create a new directory with the name of the application

5.1.1.3 Configure a virtual server for the application

Every single application has a dedicated virtual server.

5.1.1.4 Modify hostname file on the system to point to the installed application when launching the application

5.1.1.5 Add the application name to installed applications database.

Installed applications database is a flat file database using TinyDB.

5.1.1.6 Restarts the embedded web server

5.1.2 Uninstallation

This step is simpler than the installation process. It consists of four steps

5.1.2.1 Removing the application folder from the file system.

5.1.2.2 Modifying the hostname file by deleting the application name from it

5.1.2.3 Updating installed applications database

5.1.2.4 Restarts the embedded server

5.1.3 Viewing Installed Applications

The application manager reads the installed applications database and views them to the user. If the application has a favicon set, this icon will show on the GUI next to name of the application.

5.1.4 Launching Applications

In this implementation the application manager launches installed applications using Google Chrome by default. Since, hostname file is configured properly during the installation process, we just launch Google Chrome with the right flags and give it the name of the application. The browser will be directed to the local virtual server where the application files are located. For our future work, we plan to make it possible for the user to choose their favorite browser to run the applications.

Embedding a webserver is an integral part of this design. In our design and implementation, we picked *lighttpd* server [16] as our backend. Here is why we think this webserver is the right choice.

- It is a lightweight server compared to Apache webserver for example. The compressed version of the whole source code is only 2.2 MB, which makes it easier to embed in our application manager.
- It is easy to configure a virtual host for every application.
- It has a full support for php code, so web developers can continue using php and make web applications.
- It supports running code in high performance programming languages such as C++ through its fastCGI.

6. EVALUATION

This design is implemented in a Linux environment. We were able to download multiple web applications from GitHub and install them locally. We tested the design on a desktop machine running Linux Mint with Intel Quad Core processor and 8 GiB of memory.

In our evaluation of this design, we concentrated on the startup time for the application and the performance of the application itself.

We defined the application startup time as the time needed for an application to reach a state where the user can interact with it.

In this design, total startup time is the time taken to start the browser added to the time taken to load the code from the local server.

The average startup time for Google Chrome in our experiment is 882 milliseconds (about 1 second). In figure 3 we see the total startup time for the web applications in our experiment. It is worth mentioning that Audio Player and WebGL applications are written in pure JavaScript. However, Compression and Photo Editor are written in WebAssembly. The photo editor took a longer time to load because it is based on OpenCV that we had to compile to WebAssembly and load it to the page.

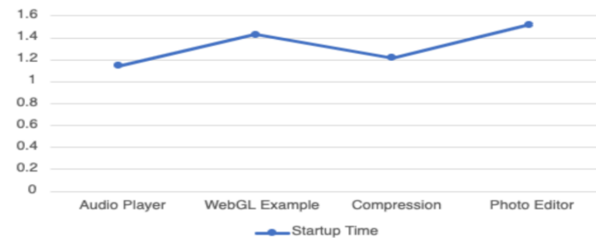

Fig 3: Startup Time for the Applications in or Experiment (seconds)

We noticed that the load time was shortened if we open the application more than once due to in browser caching. This decrease in load time is negligible since even with caching disabled, we still load the files from the local machine.

As mentioned in our goals for this design that we aim to build high performance applications that uses WebAssembly for CPU-intensive tasks. The audio player and WebGL applications were written in JavaScript. However, they use Natively implemented APIs in the browser namely Web Audio API and WebGL. In our test machine the WebGL application was able to achieve 60 FPs without any lag.

In our compression application we implemented two compression algorithms zstd and zip in both native codes written in C/C++ and WebAssembly. If figure 4 and 5 we see how WebAssembly implementation performs against optimized C/C++ code.

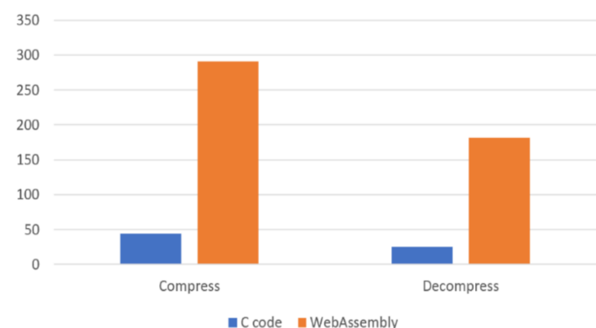

Fig 4: Zstd Compression in WASM vs C/C++ (ms)

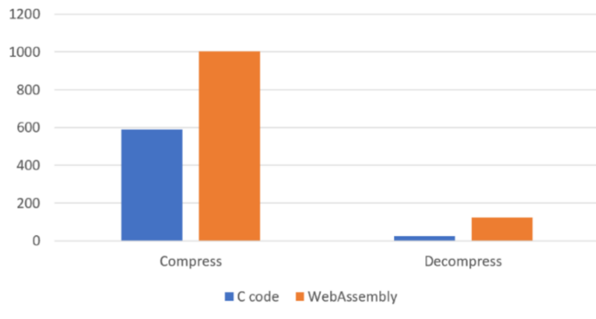

Fig 5: Zip Compression in WASM vs C/C++ (ms)

We also compared the performance of WebAssembly against native code in our image editing application. That is shown in figure 6.

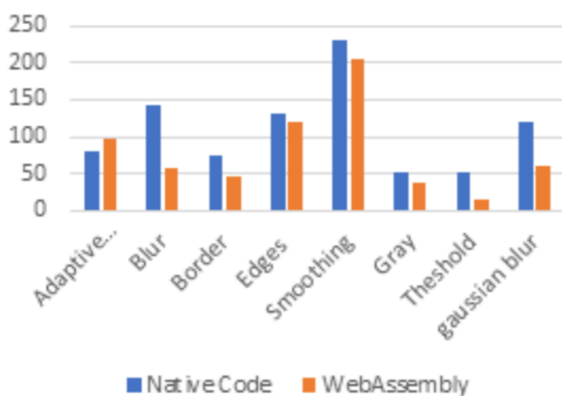

Fig 6: Photo Editing in WASM vs Native Code

Optimized C/C++ code is still better than WebAssembly in terms of performance. However, in this design developers still have the option of using C/C++ and run them using the webserver CGI. That might lead to portability problems, so we recommend using WebAssembly despite the slight performance degradation.

7. FUTURE WORK

This design has been implemented and tested on Linux. We plan to extend it to support multiple desktop operating systems such as Windows and macOS. Since we are using the browser and a lightweight webserver, this should not be hard to implement.

Unlike forementioned frameworks, this design does not force developers to use chromium-based browsers. Therefore, we plan to allow users to choose their favorite browser to be the frontend of installed applications. In our implementation we used Google Chrome, but in the next version the user will be able to change that.

For GUI in our implementation, we used Tkinter. It gets the job done, but it might not be the ideal GUI library for production. Therefore, we plan to switch to PyQt5 for a cleaner user interface.

8. CONCLUSION

In this paper we presented a novel design to build desktop applications using pure web technologies. This design does not require developers to learn a new programming language or a framework. It also relies on standards that browser vendors are committed to adhere to.

We also implemented a prototype of this design in a Linux environment. We developed application manager to install/uninstall, view, and launch web applications in the browser. Typical desktop users do not have to learn how to install a webserver to run web applications locally. They will be able to use in browser applications without the need to send data to a remote server for processing. In addition, they will not be forced to use a certain operating system just because the applications they use are not available on their favorite one.

Eliminating the barriers between different operating systems by making applications available in all of them is an ambitious dream. In this paper we presented a simple yet useful design to achieve that dream.

9. REFERENCES

[1] S. N. Dorogovtsev and J. F. F. Mendes, Evolution of Networks: From Biological Nets to the Internet and WWW. OUP Oxford, 2013.

[2] “NW.js.” https://nwjs.io/ (accessed Oct. 30, 2020).

[3] “Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS.” https://www.electronjs.org/ (accessed Sep. 21, 2020).

[4] “Performance | Electron.” /docs/tutorial/performance (accessed Sep. 21, 2020).

[5] “Figuring the Costs of Mobile App Development,” Formotus, Jun. 23, 2017. https://www.formotus.com/blog/figuring-the-costs-of-custom-mobile-business-app-development (accessed Sep. 30, 2020).

[6] D. Geradin and D. Katsifis, “Bringing an End to Apple’s Anti-Competitive Practices on the App Store: A Response to Völcker \& Baker,” Social Science Research Network, Rochester, NY, SSRN Scholarly Paper ID 3694716, Sep. 2020. doi: 10.2139/ssrn.3694716.

[7] P. Labs, “IPFS Powers the Distributed Web,” IPFS. https://ipfs.io/ (accessed Sep. 30, 2020).

[8] D. Herrera, H. Chen, E. Lavoie, and L. Hendren, “WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices,” p. 26.

[9] I. Jibaja et al., “Vector Parallelism in JavaScript: Language and Compiler Support for SIMD,” in 2015 International Conference on Parallel Architecture and Compilation (PACT), Oct. 2015, pp. 407–418, doi: 10.1109/PACT.2015.33.

[10] “WASI |.” https://wasi.dev/ (accessed Oct. 11, 2020).

[11] S. Van Acker and A. Sabelfeld, “JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript,” in Foundations of Security Analysis and Design VIII: FOSAD 2014/2015/2016 Tutorial Lectures, A. Aldini, J. Lopez, and F. Martinelli, Eds. Cham: Springer International Publishing, 2016, pp. 32–86.

[12] C. Reichert, “Microsoft Office 365 banned in some schools,” CNET. https://www.cnet.com/news/microsoft-

office-365-banned-in-some-schools-over-privacy-concerns (accessed Oct. 10, 2020).

[13] 1615 L.St NW,Suite 800Washington,and D. 20036USA202\-419\-4300 |M.\-857\-8562|F.\-419\-4372|M. Inquiries,“Apps Permissions in the Google Play Store,”Pew Research Center: Internet, Science \& Tech, Nov. 10, 2015. https://www.pewresearch.org/internet/2015/11/10/apps-permissions-in-the-google-play-store/ \\(accessed Oct. 11, 2020).

[14] “Same Origin Policy - Web Security.”\\ https://www.w3.org/Security/wiki/Same_Origin_Policy \\ (accessed Dec. 30, 2020).

[15] “Permissions updates in Android 11,” Android\\ Developers. https://developer.android.com/about/versions/11/pri\\vacy/permissions (accessed Dec. 30, 2020).

[16] “Home - Lighttpd - fly light.” https://www.lighttpd.net/ (accessed Sep. 30, 2020).