Electron and *Tauri*: Comparison of web-frameworks for building cross-platform desktop applications

Kevin Lüttge
University of Kassel

ABSTRACT

Web-frameworks like Electron provide the possibility to easily develop cross-platform desktop applications by using common web-technologies such as Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript. The increasing popularity of those frameworks is shown by various day-to-day applications namely WhatsApp, Discord or Visual Studio Code that are built with Electron. Two years ago Tauri came up, whose developers claimed it "to be faster, smaller and more focused on security than other alternative frameworks" [2]. This paper will discuss the differences and similarities between Electron and Tauri, in terms like architecture, performance or security. Therefore, a counter application is implemented for each framework to analyze and compare the development process.

1. INTRODUCTION

Cross-platform web-frameworks like Electron, Tauri or Flutter provide the possibility of using standard web-technologies like HTML, CSS and JavaScript or entire web-frameworks like Angular to develop or migrate classic desktop applications to web applications. The traditional approach of implementing desktop applications needs the developer to consider different Application Programming Interface (API) or environments of the major operating systems Operating System (OS) Windows, Mac and Linux. In fact, each application has to be implemented multiple times to adapt OS-specific requirements and produces redundant code. In context of consumer applications, the usage of frameworks like Electron allows companies and product owners to develop new target groups, but also increases the pool of potential applicants, since they provide web developers technologies and features to implement desktop applications without the knowledge of standard programming languages that are used for those kind of applications like C/C++ or Java. But also institutions or universities and their students benefit from it, because, due to the fact of cross-platform capabilities, applications can be used by many devices and decrease costs since they do not need particular hardware to be executed.

1.1 Background

Over the past years web-frameworks for building desktop applications have experienced more and more attention, which can be expressed by the number of articles related to such topics or the temporal progression of Google Trends related to this topic. This is owed to the fact of fundamental advances at web technologies like TypeScript, as an improvement of JavaScript, or frontend frameworks like Angular or React [15], which lead to a wider use of such technologies for various scenarios. A lot of every day applications used by computer scientists like Visual Studio Code or GitHub Desktop, but also applications with usage spread over industry sectors like Microsoft Teams, Skype or Discord and even Social Media applications like WhatsApp or Twitch are implemented utilizing cross-platform web-frameworks. The trend of using those applications has even grown up since the outbreak of SARS-CoV-2 in early 2020 [8] and the enlarged number of employees working from home as a result of lockdowns all over the world. This lead to an increasing number of cross-platform web-frameworks with different approaches in context of building, security or performance to provide a fast and simple way even for small development teams to use web-technologies for implementing desktop applications, since their cross-platform capabilities prevent redundant implementation of the same application for different OSs.

1.2 Motivation

As mentioned in Chapter 1.1, various cross-platform web-frameworks came up using different technologies or languages for their backend and frontend core. One of the oldest and most commonly used frameworks is Electron, which is backed up by several popular applications mentioned before, that are utilizing this framework. Therefore, Electron has become a standard in context of cross-platform development of desktop applications and almost every new framework invented is benchmarked against it like Flutter [13], JavaFX [6] or Electron [10]. Two years ago a new framework called Tauri was introduced, based on the discontent with Electrons high memory consumption or insecurity by its developers. It was designed to improve several aspects of Electron, especially in case of performance, memory usage and security [2]. Since these statements are made by the developers and thus may be subjective, this paper aims to provide an objective overview to the reader of both frameworks, Electron and Tauri. Therefore, fundamental architecture as well as the frontend and backend core of each framework is explained in detail at Chapter 2 for Electron and at Chapter 3 for Tauri. This technological knowledge is

applied by implementing a basic application which will be described in detail at Chapter 4 and analyzes different aspects of both. To obtain an objective comparison between both frameworks, Chapter 5 examines the results of previous Chapters to isolate advantages and disadvantages to the reader. At the end of this paper, the comparison results of Chapter 5 are contrasted against the statement of the Tauri developers from [2] and being discussed at Chapter 6 to provide an unbiased evaluation of both frameworks.

1.3 Related Work

An exploratory study of the utilization of different frameworks for desktop apps has worked out, that applications developed with cross-platform web-frameworks are made for various kinds of usage [18]. The authors also empirically documented that those frameworks are mostly used by developer teams with a median size of 1, which is a direct result of the amount resources classical desktop development consumes. Nevertheless, they also discovered disadvantages using such web-frameworks, like a high number of used libraries due to the fact of compensating a lack of features provided by the frameworks, but also a high ratio of platform-related issues to all issues of 20%.

The increasing popularity of web-technologies has been shown by [15]. Since they identified their native Java Swing desktop application as a bottleneck, the authors decided to replace it with a technology stack providing long term sustainability. Therefore, they used several web-technologies like AngularJS or Typescript to implement "a web-based tool for configuring experiments on the National Ignition Factory" [15]. But they also came in touch with typical problems of the JavaScript ecosystem like exit or replacement of widely used technologies forcing them to migrate from AngularJS to its successor Angular. It has to be mentioned that the authors emphasized the community support especially, in case of migration by providing tools for those cases.

Using Electron as a standard benchmark for cross-platform web-frameworks is corroborated by various papers comparing it to other frameworks like JavaFX [6], NW.js [10] or Flutter [13]. Both authors compared different web-frameworks against Electron. Although the benchmark of the authors of [6] was made with different Integrated Development Environment (IDE)s and the comparison between Flutter and Electron by [13] was based on a beta version of Flutter, both pointed out that at the one hand Electron has better performance in case of execution time and CPU consumption and on the other hand it provides more features than the compared frameworks.

2. ELECTRON

Electron originally was released as Atom Shell by GitHub at 15th July, 2013 [25] and downloaded approx. 83 000 000 times[1] The intention of the developers was "to handle the Chromium/Node.js event loop integration and native APIs" [17] for the Atom Editor. On 23rd April, 2015 it was renamed to Electron and announced that the developers want to provide a framework that allows to build desktop applications only with web-technologies. Many day-to-day apps have been implemented using Electron like Discord, Twitch or even Microsoft Teams. This has lead to an increasing community which can be expressed numerically based on GitHub Statistics [21] and is shown at Table 1:

Stars	Forks	Watching	Used by	Contributors
130 000	13 700	2 900	244 568	1 126

Table 1: GitHub Statistics of Electrons Repository

Electron combines both Chromium, an open-source browser and Node.js, an open-source JavaScript-runtime, to provide frontend-based features like rendering HTML content as well as OS-based features like filesystem access inside one framework.

2.1 Architecture

Electrons architecture fundamentally relies on Chromium, which is included in each Electron executable and has a lot common with it. As Chromium, Electron is based on a multiprocess model, containing of a single process called **main** and several processes, one for each window, called **renderer**. Figure 1 shows such a multiprocess model structure, where the `main` process creates and manages multiple `renderer` processes [11].

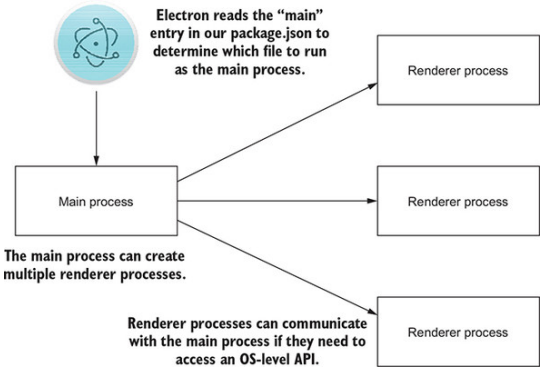

Figure 1: Multi-process Model Electron from [11, Fig. 1.7]

Main:
This is the main entry point of the application and responsible for lifecycle management like starting or quitting the app. The `main` process uses so called `BrowserWindow` module to create and manage each `renderer` process which is loading web pages into it. Since only the `main` process is running inside a Node.js environment, it is the sole part of the application that can import Node.js modules using `require`. This forces each `renderer` process to communicate with the `main` process, if they want to consume system APIs for purposes like saving files or opening dialogs. This communication is called Inter-Process Communication (IPC) and Electrons implementation of it will be discussed in detail at Chapter 2.1.1 [22, 11].

[1] According to `https://npm-stat.com/charts.html?package=electron&from=2013-07-13&to=2022-07-30`

Renderer:

A `renderer` process is responsible for rendering web content by loading web pages into it and presenting them to the user at the browser window. Additionally, JavaScript code can be loaded and executed inside a `renderer` process. Each `renderer` process can be created or destroyed by the main process using the `BrowserWindow` module as mentioned before. This leads to the fact, that `renderer` processes are isolated from each other following the Chromium principles of a multiprocess model and is reasoned by limited affection of faulty or malicious code on the entire app. The `renderer` processes are only able to communicate between each other indirectly via the `main` process or by using `MessagePorts`. [22, 11].

2.1.1 Inter Process Communication

As already mentioned the `main` and `renderer` processes are only able to communicate using IPC. Therefore, Electron

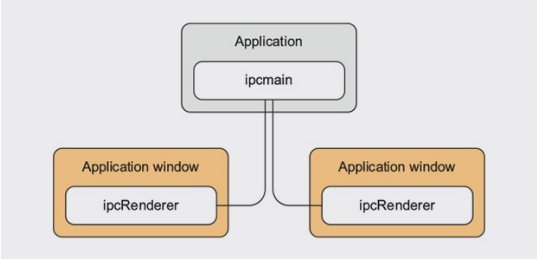

Figure 2: IPC Communication of Electron [10, Fig. 6.5]

provides two modules, one for the `main` process, called `ipcMain` and one for `renderer` process, called `ipcRender`. Figure 2 points out the process of exchanging data between two or more processes. Both of them are Node.js `eventEmitter` modules and capable of executing asynchronously and synchronously communication either uni- or bidirectional. It should be noticed, that this kind of IPC communication is only for `renderer` to `main` and vice versa, which can be seen in Figure 2. A communication between different `renderer` processes can be archived either indirectly using the `main` process or directly with `MessagePorts`. The advantage of using the direct variant is, that the workload of the `main` process can be reduced by using a new, hidden `renderer` process, which serves as a worker for the web content `renderer`. A `MessagePort` can be seen as a communication channel between two `renderer` processes. It is once defined inside the `main` process which informs the `renderer` processes about belonging `MessagePorts` from other processes and establishes the connection. After that, the actual communication does not rely on the `main` process anymore. This will reduce workload for the `main` process in case of message forwarding as well as outsourcing heavy workloads to worker processes [22].

2.1.2 Context Isolation

Electrons multiprocess model also intends distinct purposes for each process. As described before only the `main` process has access to node modules. In contrast to that only the `renderer` process has access to HTML Document Object Module (DOM)s. Since using just IPC represents a major security issue, Electron introduced a feature called **Context Isolation**. This splits the logic of a `renderer` process into two different contexts. On the one hand the `renderer` process as already described and on the other a so called `preload script`. The `preload script` is attached to the `main` process at the creation of the `BrowserWindow` module. It has access to both node modules and HTML DOMs at its own context and will be executed before the `renderer` process loads the web page into it. Although the `preload script` and the `renderer` process do both own a window object, which provides the displayed browser window, it is not the same object since they are both running at different contexts. The `preload script` consists of a `contextBridge` module which is responsible for safely exposing selected properties of the `main` process to the `renderer` and vice versa. Inside this module an API can be defined for providing access with IPC objects to resources of different processes [22].

2.2 Frontend

The frontend part of an Electron-based application consists of the Chromium content module and provides all the core features that are required to render HTML content or access web-APIs. It is important to notice, that the content module is not equal to the Chrome web-browser since the web-browser wraps around the content module, but provides several features, the content module does not provide, like managing bookmarks, spellchecking, safe-browsing or securely saving passwords. The content module itself includes two parts. A browser engine called **Blink** and a JavaScript engine called **V8** [11].

Blink:

Blink serves as a browser engine inside the Chromium content module and therefore is responsible for translating HTML documents to the actual view, a user gets presented. Originally Chromium used Webkit as browser engine, which was developed and maintained by Apple, but fundamental disagreements between Apple with its restrictive policy and Google led to the fact, that Google forked the Webkit engine and uses it as a grounding for their own engine [7]. It relies on the same architecture as mentioned at Chapter 2.1, whereas Blink is running inside a `renderer` process and uses one main thread and multiple worker threads that do the layout calculations [16].

V8:

V8 is written in C++ and responsible for compiling and executing JavaScript code as well as memory management like allocation or garbage collection. It can be seen as the executing background part of the content module, which transforms JavaScript code into C++ code, which in turn is translated into machine-readable byte-code, but it also provides the possibilities for developers to write own C++ applications that can expose their functions to JavaScript code to add new features or improve performance of execution [3].

Summarized the content module provides all features to Electron that are necessary to develop a classical browser application. But since classical browser applications are restricted by the OS for example in case of filesystem access,

the use-cases are limited. This is the reason, Electron has combined the content module with the Node.js runtime.

2.2.1 Backend

As backend for all Electron-based applications the Node.js runtime is used. Node.js is a framework allowing developers to implement server-side applications using JavaScript, and as the Chromium content module of Chapter 2.2 it utilizes V8 to execute JavaScript code, but also provides more functionalities like the mentioned filesystem access or the possibility of extern module import. Furthermore, a package manager, called Node Package Manager (NPM), with "more than one million packages"[2] is included. This amount of packages is a result of an increasing popularity Node.js experienced since its release, providing the capabilities to implement a wide-range of applications. The reason for utilizing Node.js runtime as a backend for all Electron applications is its privileges. At traditional web-applications, the client-side is restricted through the OS to consume or request data from a third party API. Because of that, the client has to request the server, which then is consuming the third party API. Since Node.js has these privileges, the detour at the server-side falls away, guaranteeing Electron applications these accesses at the client side [11, 10].

3. TAURI

Tauri was first released at 31st December, 2019 [14] and downloaded approx. 58 000 times[3]. It resulted as discontent of Electron from the Tauri developers especially in case of resource consumption and security due to uncontrollable dependencies [2]. They criticized the enormous resource consumption even of the simplest applications as well as the fact, that Electron does not have control over their dependencies. If Chromium encounters a zero-day-exploit and releases a patch, Electron has to include this patch and also release a newer versions. This leads to the fact, that users of an Electron-based application have to update their Electron version to close this security issue. This timespan, between first fix of Chromium and the users update of the application, is a high vulnerability against potential attackers. In this regard, the developers also criticized the power of the privileges Electron applications require, allowing attackers to have access to the entire hard drive of the user for example [2]. But like Electron, Tauri experienced increased attention kept in mind that the first stable version has been released this year, which also can be expressed numerically based on GitHub Statistics [20] shown at Table 2:

Stars	Forks	Watching	Used by	Contributors
48 200	1 200	403	3 200	182

Table 2: GitHub Statistics of the Tauri repository

But unlike Electron, Tauri uses the self developed core, called Tauri, which is written in Rust in combination with Webview Rendering Library (WRY), which serves as the rendering library.

[2] `https://www.npmjs.com/`

[3] According to `https://npm-stat.com/charts.html?package=tauri&from=2019-12-31&to=2022-08-12`

3.1 Architecture

The architecture of Tauri is very similar to Electrons multiprocess model. Tauri also relies on a main process, called **core** process and multiple rendering processes, called **webview** for performance as well as security reasons. Figure 3 shows the basic architecture of Tauris multiprocess model whereas each `webview` process is managed by the `core` process.

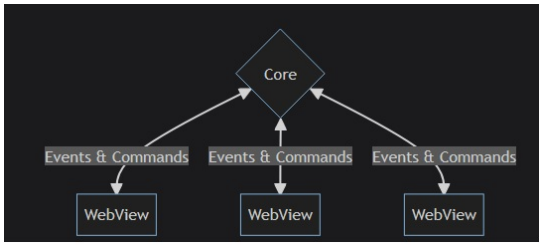

Figure 3: Multi-process Model Tauri from [2]

Core:
This is the main entry point of the application and the only place where full access to the operating system is provided. The `core` process uses this privileges to create and manage the windows of the application. But it is also the only point, where the communication between processes is going through, allowing the `core` process to manipulate or observe IPC messages. Unlike Electron, `webview` processes are not able to communicate directly between each other, increasing the security since each message has to pass the `core` process and can be discarded if suspicious or unwanted calls are made. Additional the `core` process is responsible for global scoped cases such as database access or managing application or OS-specific settings that affect the windows. Summarized it serves as a centralized management and control point, where the application as itself is maintained and sensitive data is kept to be hidden from the `webview` processes [2].

WebView:
A `webview` process renders the HTML content of an application using the WebView libraries of the current OS. Since this library is not included into the final executable but linked at runtime, its appearance differs depending on the OS the application is running at. This reduces the size of the executable since the part where the actual rendering takes places is shifted from the application to the OS but also forces developers to consider and keep in mind the different OSs [2].

3.1.1 Inter Process Communication

For communication between different processes Tauri also uses Inter Process Communication similar to Electron. In contrast, Tauri forces developers from the beginning to use the Asynchronous Message Passing (AMP) paradigm, whereas Electron released this feature at version 14 and only recommends it. The main advantage of AMP is, that direct function access is denied and thus all communication has to pass the `core` process. Since it is able to observe the content of messages, it can decide which message will be forwarded

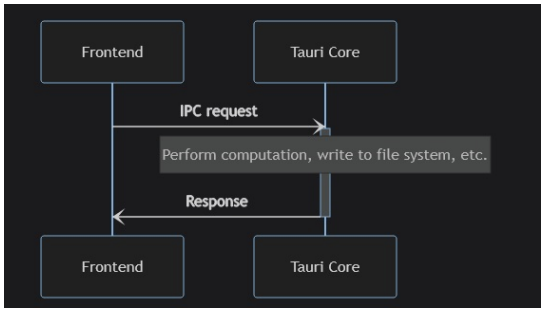

Figure 4: IPC of Tauri [2, Figure 1–3]

and which will be blocked like malicious requests. Communication can be either unidirectional using events, which can be emitted by both `core` and `webview` processes to inform the event recipient but without any response, or using commands which are bidirectional IPC messages but can only be emitted by the `webview` processes to invoke functions that require access to the operating system like it is shown in Figure 4. The `core` process itself is not able to emit commands to the frontend preventing potential malicious code segments inside the `core` process affecting the `webview` processes [2].

3.1.2 Context Isolation

In contrast to Electron Tauri uses different patterns for isolating communication between the `webview` processes and the `core` process and potential critical API calls. They are called Brownfield and Isolation Pattern, whereas the default pattern, that can be configured inside the `tauri.conf.json` is the Brownfield pattern.

Brownfield Pattern:
The Brownfield pattern can be seen as a design pattern to ensure interoperability between new implemented and existing software. This pattern does not categorize software as legacy software but as current state of the art and software developed following this pattern tries to coexist and consider existing software as much as possible. This requires a deep knowledge of the existing software and also can result in re-developing significant parts of the existing software when tried to enhance new features. Tauri uses this pattern as standard and explains that it tries to be as compatible as possible. But unfortunately Tauri does not explain in detail how they are implementing this pattern and how this helps to avoid malicious frontend calls to the Tauri core [2].

Isolation Pattern:
The Isolation Pattern can be seen as an interposed instance between the IPC handler and the processes. This instance is providing a sandbox, called Isolation application, which is trusted and secure JavaScript code embedded into an `<iframe>`. The IPC handler passes its message to the Isolation application, where it is executed and may be modified or verified. After that it will be encrypted, passed back to the IPC handler and forwarded to the `core` process, where it will be handled as normal [2].

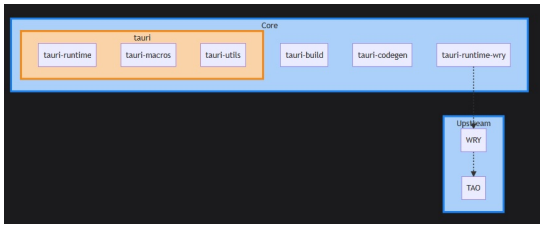

Figure 5: Core Ecosystem from [2]

As it can be seen in Figure 5 Tauri splits its framework into two main components. The Core, containing several modules providing API access, utilities and the runtime and be can be seen as the backend part of the framework. The Upstream component, serving as the frontend part, containing the two self-developed Rust crates WRY and TAO, proper name, which are providing libraries for creating browser windows and rendering the WebView and its content.

3.2 Frontend

As mentioned above, Tauris frontend relies on its own implementation of WebView instead of the entire Chromium content module for rendering HTML content. This makes it much smaller than Electron applications since their libraries WRY and TAO are using the exising web engines of the three major operating systems Linux, macOS and Windows instead of shipping an entire browser.

TAO:
TAO is a cross-platform library written in Rust and used for creating and managing application windows. It was forked from the Rust crate `winit` since the developers wanted to enhance more desktop features than the original crate, like menu bars or a system tray. This library ships with an entire event loop that can be emitted by windows like resizing or key interactions and is included at the second frontend library WRY [1].

WRY:
WRY is the main library of Tauri applications and responsible for rendering the content of the OS-specific webview and based on the webview library [23]. It acts as an interface between the Tauri core with its low level webview drivers and the webview technology of the current operating system, providing a unique abstraction layer for rendering HTML content. Since it re-exports TAO and its event loop to guarantee access to OS-specific web engines the appearance of the same application may differ on each operating system, depending on the underlying web engine [4].

3.3 Backend

The backend or core of Tauri contains several components, whereas some of them are summarized and called `tauri` to emphasize the main part of each Tauri application including runtime or the Tauri API. The decision to write the entire backend in Rust was made because of the ownership feature in Rust which provides a set of rules for memory management and avoiding a garbage collector like Java or explicit memory allocation like C [2]. It can be seen as an approach of trying to be more comfortable for the developer

as an explicit allocation but also to force the developer to consider his/her memory handling to implement an efficient application [12]. This set of rules is checked at compile time and if one of them is violated, the program will not compile. Beside the `tauri` components there are also additional components included by the core like system-level interactions for the upstream crate WRY . One of the major advantages of using Rust for the Tauri backend is as already mentioned its ownership rules, avoiding security issues but also the execution time of Rust code which was intended to obtain a similar performance as C++ [12]. Furthermore, Rust experienced a significant attention resulting in the adoption of it by big companies like Google or Facebook which assumes an increasing position.

4. IMPLEMENTATION OF A COUNTER APPLICATION

Regarding Chapter 1.2 this Chapter describes the underlying work for this paper and will guide through the entire development process of Electron and Tauri applications. To obtain comparable results, the methodology of comparison will be explained shortly. For this paper a basic counter application has been implemented in both Electron and Tauri using just HTML,CSS and JavaScript, although both frameworks are supporting most common frontend frameworks like Angular, React or Vue.js. This decision was made because the paper focuses on presenting the differences and similarities of each framework to the reader and thus using complete frameworks will both blow up the entire application and not concentrate on the essentials. The counter application consists of a simple number serving as counter which is displayed and two buttons providing the possibility to increment or decrement the counter. Both buttons will have an event listener, that sends IPC messages to the backend and as response gets the new calculated counter value which then will be displayed to show the entire communication chain of each framework. Therefore, each application will contain the same HTML content that is displayed and only the API calls of the frameworks will differ, depending on their architecture. To avoid unnecessary distortion of measurements only the fundamentals of each framework are included without any additional helper libraries for better styling or further functionalities.

4.1 Methodology

For benchmarking the applications in case of build time GitHub Actions are used. Thus, each project has its own GitHub repository with a workflow.yaml defining the Actions' workflow. Each Action will run on the three major operating systems macOS, Windows and Linux, set the prerequisites for the framework respectively and use the recommended build tool, `electron-forge` for Electron and the `tauri build` command for Tauri. This will result in three jobs for each project, whereas the build time for each framework on each OS will be measured, since usual prerequisites are installed once and thus not measured. Time of execution is measured using hyperfine[4] which is a command line benchmark tool. Therefore, the executable of each framework will be started by command line, with hyperfine switched in front. To see the difference between a cached and uncached execution, the first run does not have any warmup actions and so prevent the OS from loading the application into the filesystem cache. This results in following command:

```
# hyperfine -runs 10 -warmup 2 'start <executable>
```

To gather meaningful data, hyperfine will run 10 instances of the executable to obtain a min-max range as well as a mean execution time with its standard deviation.

For memory consumption the python library memory-profiler[5] will measure the memory consumption of each executable over a timespan of 60 seconds, enough to allow interactions with the application as well as getting from startup to idle state. Since the applications rely on multiprocess models and may also spawn child processes, these are recorded too, to gather trustful memory consumption measurements. To archive this measurement following memory profiler command is used:

```
# mprof run -include-children -multiprocess -timeout
60 <executable>
```

It is important to notice that Tauri comes with a single executable file, whereas Electron ships with an installer which has to be executed first in order to get the actual executable application running. All Electron measurements will use this installed executable as foundation.

Except build time for different operating systems which will use GitHub Actions, the measurements of execution time, memory consumption or binary size are run on Windows 11. The implemented application is based on Electron Version 19 the newest stable release at time of writing this paper respectively Tauri version 1.0.2.

4.2 Development

The development process is following the best practices section of each framework [22, 2] to avoid deprecated or inefficient calls. Implementation is done with usage of Visual Studio Code, a lightweight IDE commonly used for web development and also recommended by both Electron and Tauri.

4.2.1 Prerequisites

Since Electron mainly relies on Node.js, this as well as the actual Electron framework are the only dependencies that are needed to start developing. It should be noticed, that Electron is installed as dev dependency since packaged applications are shipped with an Electron binary and therefore are not needed to be defined as production dependency [11]. Although they are not used, since the application is based on just HTML CSS and JavaScript, Electron provides several tools for generating a project with fundamental boilerplate code for most common web frameworks like Angular or React.

Tauri in contrast needs several more prerequisites to start developing which includes WebView2, the actual web engine for Windows that is used by Tauri as well as Rust, what on the other hand pre-conditions Microsoft Visual Studio C++ Build tools. Since Tauri consists of two subprojects, the Rust Core and the Frontend it provides a tool for scaffolding boilerplate projects using the most common webframeworks as well as just plain HTML CSS and JavaScript (called Vanilla.js by Tauri), whereas Node.js as prerequisite is implied too. To initiate the Rust project, once the entire project is scaffolded, the CLI module of Tauri is required.

[4] `https://github.com/sharkdp/hyperfine`

[5] `https://pypi.org/project/memory-profiler/`

This tool will generate a minimal Rust project set-up to use Tauri with the selected web-framework and specify the location of all the belonging web assets.

Once each project is initialized a basic HTML file combined with simple CSS styling is created to specify the actual appearance of the application. The main JavaScript file of the frontend is specified inside the `package.json` of each framework. As mentioned before, two buttons will be added to increment or decrement the counter for invoking a basic IPC chain. Therefore, each button has an event listener which will listen to click events and call the appropriate IPC handlers.

4.2.2 Implementation - Electron

Following best practices the Electron framework is divided into three JavaScript files, whereas the main process (`electron.js`) represents core process as described in Chapter 2.1. This file contains the actual browser creation which will spawn a new renderer process with the corresponding preload script as well as specifications of the content that should be loaded inside the window. But also required handlers of the `ipcMain` module including the logic are implemented here. The second JavaScript file is the preload script that contains the `contextBridge` and `ipcRenderer` modules of Electron and exposes the API to the renderer process. This preload script is linked to the main process at the definition of the browser window, that is created by the `electron.js` file. The third JavaScript file is the renderer script, that is responsible for adding basic interaction processing that can occur.

```
1 inc_btn.addEventListener('click', async ()
      => {
2     counter.innerText =
3         await window.electronApi.increment
              (document.getElementById('
              counter').innerText)
4 })
```

Listing 1: Excerpt of render.js

```
1 contextBridge.exposeInMainWorld('
      electronApi', {
2     increment: (param) => ipcRenderer.
          invoke('incrementChannel', param)
3 })
```

Listing 2: Excerpt of preload.js

```
1 ipcMain.handle('incrementChannel', async(
      param) => {
2     return parseInt(param) + 1;
3 })
```

Listing 3: Excerpt of electron.js

Once a button is pressed a click event will be emitted and the code shown at listing 1 will be executed. Inside the event listener the related function of the API, that is exposed by the preload script and shown at listing 2, will be called. This will invoke the `ipcRenderer` module which takes the parameters of the `increment` function and send a message using the defined channel, at this case `incrementChannel` and wait for the response of the handler of the `ipcMain` module at listing 3. Since the handler is inside the `electron.js` which is the main process of the application, this is the place where node modules can be imported and thus the only way for renderer process to use them. The handler will return the new calculated passed parameter back to the `ipcRenderer` which then will be set as new counter value as it can be seen at listing 1. Addressing the difficulty of implementing an application with Electron is has to be said that the entire app can be written with JavaScript,HTML and CSS which may have a great impact on small teams of web-developers that are already familiar with those web technologies. The framework itself is well documented including examples that cover of most use cases and constantly updated to follow the recommendations of the newest releases by the maintainers. Since Electron uses Node.js as runtime it benefits from its huge community providing libraries for almost any scenario. Nevertheless, since Electron has released up to 19 stable release versions, there has changed a lot over the years, causing developers to update their applications constantly, which in worst case could result in reimplementing the entire process communication to migrate their codebase.

4.2.3 Implementation - Tauri

Once the two subprojects of the Tauri application are initialized the same frontend as described in Chapter 4.2.2 will be added to the specified web assets folder, containing the HTML file, a CSS stylesheet as well as the JavaScript file for processing interaction events, at this example the EventListener of both buttons.

```
1 incBtn.addEventListener('click', function
      () {
2     invoke('inc', {cnt: parseInt(counter.
          innerText)}).then((response) =>
          counter.innerText=response)
3 })
```

Listing 4: Excerpt of index.js

```
1  #[tauri::command]
2  fn inc(mut cnt: i32) -> String {
3    cnt+=1;
4     return cnt.to_string();
5  }
6
7  fn main() {
8    tauri::Builder::default()
9      .invoke_handler(tauri::
           generate_handler![inc])
10     .run(tauri::generate_context!())
11     .expect("error while running tauri
           application");
12 }
```

Listing 5: Excerpt of main.rs

The core process as described in section 3.1 is created by the `main.rs` file of the Rust project inside the main function of listing 5. Inside the main method the default function of Tauris Builder structure is executed which will set WRY as runtime for the application. The `invoke_handler` method will set up the passed handlers of the application that will be responsible for IPC. Inside the `run` method, that will execute the passed context, the actual context is generated by the `generate_context` method, that will read the config file,

which is the `tauri.conf.json` by default and create the context of the application. To indicate handlers that will be executed once the frontend makes an IPC request to the core, Tauri uses the `#[tauri::command]` makro for commands and `#[tauri::event]` for events respectively. As it can be seen at listing 4 once a button is pressed, the implemented event handler will be called. This will call the `invoke` method of the Tauri API which can either be imported as NPM package or set to global inside the config file. The method takes two parameters whereas the first defines the name of the command that should be executed, similar to the channels of Electron and the second parameter is containing all data that should be passed to the command method and is returning a promise which then can be processed. Since Tauri is using a protocol similar to JavaScript Object Notation - Remote Procedure Call (JSON-RPC) the passed data has to be serializable. After the IPC request has reached the core, the corresponding method will be executed which can be seen in listing 5. The command takes an integer as parameter, increments it and then returns a string, that will be wrapped into a promise. After the invoke method returned the promise which contains the processed data of type string it will update the counter. Referring the complexity and difficulty the reader should keep in mind that implementing a Tauri application nowadays requires knowledge in Rust, since the backend and therefore the process communication can only be implemented in Rust. Although the Tauri developers announced plans to provide the support of different languages for the backend at the time, this paper is written, it only supports Rust as backend language [2]. The documentation of Tauri tries to follow the same schema as Electron but is often indistinct and ambiguous or does not provide any detailed information of subjects like the Brownfield pattern of Chapter 3.1. Especially the choice of using Rust as backend the libraries provided by Rusts package manager cargo are limited in contrast to Node.js and may result in writing own libraries to add required functionalities more often.

4.3 Performance

Once the applications are implemented the build process of each framework is invoked to create an executable which is the foundation of performance measurements. The build time for each application is measured using GitHub Actions, whereas it has to be mentioned, that this will be done on completely bare metal runners, that did not cache anything to obtain more comparable measurements since each framework may cache at different scales. After the executables are created the performance of both are measured using the techniques described in Chapter 4.1.

4.3.1 Build Time

Table 3 summarizes the results of the GitHub Actions for each operating system and the binary size. It is clearly visible that building a Tauri app requires much more time than Electron applications spread over the operating systems. This can be explained with the complex process of Rust compilation like incremental compilation techniques or time-consuming code analysis [24] combined with the absence of caching since the GitHub Actions were executed on bare-metal systems. This is owed due to the fact of providing comparable results and may not occur at every-day development where applications are build constantly to test features and therefore cached to reduce compile time.

Building time in seconds				
Framework	Operating System			Size [KB]
	Windows	Ubuntu	MacOS	
Tauri	863	616	389	6 924
Electron	115	121	44	145 361

Table 3: Building time of the counter-app executables of Electron and Tauri frameworks at different OS using GitHub Actions

According to Table 3, the binary size of the compiled executable amounts to 6 924 kB for the Tauri application respectively 145 361 kB for Electron which is a direct result of the architecture of both frameworks described in Chapters 2.2 and 3.2. Electron ships the executable with an entire web browser, whereas Tauri makes use of the system specific web engine and therefore is able to reduce its size by a factor of approximately 21. This may be an advantage or disadvantage depending on the developers or customers point of view, since the different web engines of each system result in slightly different appearances of the application.

4.3.2 Memory Consumption

Figure 6 and Figure 7 show the entire memory usage of the implemented applications including their child processes. Although the Electron application consumes more memory at startup, causing a peak of approximately 400 MiB at Figure 6 it decreases its memory consumption up to approximately 290 MiB, whereas the Tauri Application is constantly at a consumption of roughly 320 to 330 MiB. Both figures also point out that Tauri has more processes involved into the application than Electron. Those graphics stay in contrast to the actual benchmarks provided by Tauri[6] and thus only the benchmarks for Electron are made public it is not possible to explain the discrepancies between both frameworks objectively.

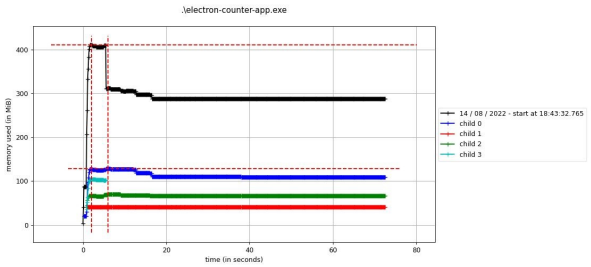

Figure 6: Memory consumption of Electron executable obtained from memory profiler

4.3.3 Execution Time

The measurements of `hyperfine` are shown at Table 4 for Electron respectively Table 5 for Tauri. The measurements of the mean values include their standard deviation based on ten measurements and caching describes the warmup option that is added to the second run of hyperfine. The high

[6] `https://tauri.app/about/benchmarks/`

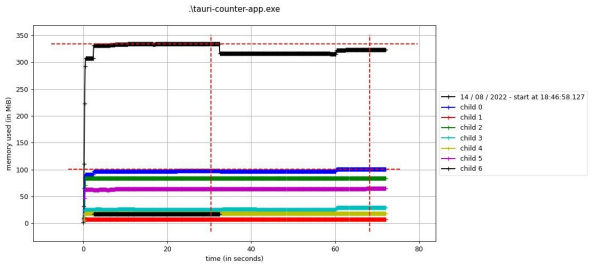

Figure 7: Memory consumption of Tauri executable obtained from memory profiler

standard deviation of both Table 4 and Table 5 at the **Without caching** row can be explained with the max values of the same row. The max value is more than 2 standard deviations away from the mean value meaning it is not a probable execution time, but it also shows the effect of filesystems caching. To obtain more reliable and undistorted measurements the warmup option of hyperfine is included at the values of the **With caching** row showing the results of execution time after the application is cached. Both tables point out, that the execution time of the Tauri application is faster than the Electron application at all measured values even if the mean values of Electron and Tauri only slightly differ. This is reasoned by Tauris' usage of Rust as backend which has a performance similar to C++ [9] and thus is more performant than JavaScript [19].

Execution time of Electron [ms]			
	Mean	Min	Max
Without caching	17.3 ± 23.3	6.5	82.9
With caching	18.0 ± 14.8	7.5	49.6

Table 4: Execution time of the Electron executable measured with hyperfine

Execution time of Tauri [ms]			
	Mean	Min	Max
Without caching	17.1 ± 10.5	5.8	35.4
With caching	15.1 ± 7.9	5.2	29.0

Table 5: Execution time of the Tauri executable measured with hyperfine

5. SUMMARY

To summarize the content of Chapters 2 and 3 as well as the measurement results of Chapter 4 both frameworks will be put in contrast to point out differences or similarities of each.

5.1 Differences

As described in Chapters 2 and 3 both frameworks rely on different technologies to build up their backend or to render the content of a web page. Electron uses the Chromium web browser which is also shipped with the compiled executable resulting at higher binary size than Tauri. Tauri relies on the underlying operating systems web engine and outsources the rendering to that engine to save memory. But this causes developers to take care of the different operating systems and their specific requirements, whereas Electron solves this problem with the Chromium web browser. Although both frameworks provide cross-platform capabilities and support the three major OSs, only Electron provides full cross-platform capability meaning that developers are able to implement an application only once and run it on each OS without differing visual appearances of the application. But it also has to be mentioned that an Electron executable ships necessary libraries as extern parts, whereas the Tauri build tools are packaging required dependencies and libraries into a single executable file. This results at higher security since only a successful decompilation opens gateways to potential attackers, whereas Electron libraries could be compromised without decompilation and thus corrupt the executable. Another important subject that both frameworks differ is the used language. Electron applications can be written completely using JavaScript providing easy access to web-developer, whereas development using Tauri requires knowledge with Rust, which has a high learning curve due to its complexity compared to JavaScript. Nevertheless, Chapter 4.3.3 points out that the performance of Tauri applications is slightly better in case of base applications. The reader has to notice, that all measurements were done using a simple counter application without any cpu or memory intense workloads and thus the better performance of the Rust backend compared to JavaScript might have a bigger impact on execution time. Another difference of both frameworks is at the subject of build time since Rust, in spite of improvements has a high compile complexity [24] although this is mostly avoided by caching and may not have such a big impact at day-to-day development. The Rust backend itself points out another difference between Tauri and Electron since it guarantees memory and thread safety and its model of ownership forces the developer to consider the efficiency of his/her application [12]. At points of library support Rusts own package manager cargo is limited in terms of the amount of provided libraries compared to NPM. This results in the fact that some functionalities are not provided by libraries and thus have to be implemented by the developers on their own. But also security aspects differ between both frameworks, although both are concentrating the main access to OS-specific operations at the main or core process. Especially Tauris' Isolation Pattern has to be highlighted, since it allows to execute suspicious frontend calls at an isolated, trusted sandbox environment before it is forwarded to the main core. This helps to avoid malicious software getting access to the core or obtaining privileges it should not have. Since Tauri applications are running serverless, in contrast to Electron, there is no possibility for potential attackers sniffing the network traffic between the frontend and the backend. But also each communication between Tauris' backend and frontend is encrypted providing additional complexity to monitor the data that is exchanged [2]. Another security aspect both frameworks differ are their dependencies. Because Electron uses NPM for managing libraries and dependencies of the application, it is up to the developer to update them and therefore is a possible open gate for potential attacks like the zero-day exploit of Log4J [5]. Although this could also happen with cargo, the features of Rust itself prevent most of the possible attacking space that may occur at NPM packages.

5.2 Similarities

Although both frameworks differ at various points especially in case of security and performance they also share similarities. Both are supporting most common and modern web-frameworks like React or Angular by providing boilerplate generation tools that create basic projects with a ready-to-use environment where the frameworks are set up to work with Tauri respectively Electron. Electron as well as Tauri use the multiprocess model as their fundamental architecture, whereas, as pointed out in chapter 2.1 and 3.1 a main process is used as entry point to the application, which is able to create and manage new browser window processes or enables communication capabilities or system-specific access. At least for basic applications they also share a just slightly different execution time. Both frameworks allow developers to decide which parts of their API are exposed to the different processes, although this is done in different ways, Electron with its preload script and the contextBridge and Tauri by simply prevent each part of the API that is not used being packaged. Despite the fact both frameworks use slightly different implementations for their IPC, Tauri implementing a protocol similar to JSON-RPC and Electron is implementing the HTML standard Structured Clone Algorithm (SCA) [22], the communication always passes the main process either directly or indirectly as message broker. But unlike Tauri, Electron provides the possibility to invoke two-way IPC request from the main process to the renderer processes, whereas Tauri allows commands only be emitted by the frontend, limiting the impact of a corrupted main process.

6. CONCLUSION

Reminding the intention of the developers of Tauri mentioned at section 3 and contrasting them to the measurements discussed in section 4 it can be determined that some aspects like execution time or security of Tauri applications compared to Electron are accurate. Tauri applications have a more efficient performance in aspects of memory usage or execution time, due to the fact of using Rust as backend core. Nevertheless, some benchmarks provided by Tauri, especially in case of memory consumption could not be reproduced or investigated further since those are not provided public. But as mentioned in section 5 Tauri compared to Electron does a step towards the traditional web development, where system-specific requirements have to be considered and therefor at worst case an application for each operating system has to be implemented. Although this might need only small changes on the entire code base redundant code follows from this. Tauri has experienced much attention since its first release but at the current state is not suitable for big applications in production mode. This statement is based on different aspects. First, the documentation is difficult to understand resulting in paraphrases or one-sentenced explanations of entire architecture aspects like their design patterns, than actual explaining them. This results in uncertainties how the Tauri framework is doing tasks in detail. Especially the security section is rather describing possible threats that can occur during development than explaining their implemented solutions or how they affect existing code. Although documentations do not have to be objective, the Tauri documentation often compares parts of it against Electron applications even if some of them are referencing old and deprecated modules like ASAR files [2]. Second, the knowledge of Rust required to implement more complex applications that need access to the Rust core. Electron can be written by just using JavaScript and thus lowers the bounds for small web-development teams, since JavaScript is widely used at this branch. For those teams, implementing a Tauri application will result in learning a complete new programming language and in worst case if they do not have any knowledge in comparable languages like C++ learning complete new programming patterns like Object Oriented Programming (OOP) principles or memory management. This consumes more resources during the development process and thus will lead to an increasing timespan between the initial development and a release of the application. Although Tauri has claimed to face this issue and announced to provide a backend that can be written using different languages, it is neither implemented nor scheduled yet. Several standard web techniques and functions like downloading or native menu items are not implemented too. This will cause developer teams add new functionalities to their applications since they are not provided by Electron yet. Third the community itself as well as the support of the developers, since Tauri and especially Rust suffers from a lack of functionalities and Features that are provided Node.js or Electron. Since Electron is used by many well-known companies like Microsoft with their Teams or Visual Studio applications or Discord and their same named application, this impacts the interests of improving the framework as well as the popularity for small development teams.

Summarized Tauri improves some problems Electron has to deal with due to its architecture and underlying frontend and backend parts but is not suitable option for implementing applications that are aimed to be provided to millions of users or small development teams that do not have the resources that are required to implement Tauri applications. As it was pointed out by the authors of [18] development teams using Electron have median number of 1 core developer. Using Tauri will result increasing complexity for those core teams although the background of the developers was not determined by the authors depending on the knowledge of system-related programming principles. Thus, the release time of applications or new features at least at the beginning of projects will shift further. This could affect developers that or even companies that want to obtain a balanced mix between release intervals and invested resources. It is a complex framework compared to other common cross-platform web-frameworks for implementing desktop applications but provides the possibility to develop efficient and small-sized applications compared to Electron, although they are currently limited by the Rust core.

6.1 Further work

Since this paper is based one the first stable release of Tauri, the topic has to be observed in the future, especially if the developers are improving their framework and documentation considering the requirements of the community. This could result in further performance comparisons between both frameworks or the productive operation of applications build with Tauri, especially with applications requiring intensive cpu workload or complex calculations. It also needs further research or observation how the developers of Electron are dealing with the issues, the Tauri developers have mentioned in the future, including investigation of devel-

opment that are using Electron if they want to migrate to Tauri or respectively as well as the reasons for that. Since Tauri is a framework with a new approach of using a system-related programming language of developing cross-platform desktop applications, further investigation in terms of how this affects establishing Rust especially as web-development language is necessary. Although the resource consumption of Electron is often criticized, there is no scientific publication how this influences the consumer choice of using those applications, since modern desktop computers do not suffer from a lack of memory in general or how this affects companies in their decision to migrate from Electron to Tauri.

7. LIST OF ABBREVIATIONS

HTML Hypertext Markup Language

CSS Cascading Style Sheets

API Application Programming Interface

WRY Webview Rendering Library

AMP Asynchronous Message Passing

JSON-RPC JavaScript Object Notation - Remote Procedure Call

IDE Integrated Development Environment

SCA Structured Clone Algorithm

OOP Object Oriented Programming

NPM Node Package Manager

IPC Inter-Process Communication

OS Operating System

DOM Document Object Module

IDE Integrated Development Environment

8. REFERENCES

[1] Tao. `https://github.com/tauri-apps/tao`. [Accessed 31-Aug-2022].

[2] Tauri blog. `https://tauri.app`. [Accessed 30-July-2022].

[3] V8 documentation. `https://v8.dev/docs`. [Accessed 13-Aug-2022].

[4] Wry. `https://docs.rs/wry/latest/wry/`. [Accessed 31-Aug-2022].

[5] Kritische schwachstelle in log4j veröffentlicht (cve-2021-44228). Technical report, Bundesamt für Sicherheit in der Informationstechnik, 2021.

[6] A. Alkhars and W. Mahmoud. Cross-platform desktop development (javafx vs. electron).

[7] Leo Becker. Google-chrome-entwickler: Apple hält das web zurück. *heise online*, 2019. [Accessed 13-Aug-2022].

[8] Alexander E. Gorbalenya, Susan C. Baker, Ralph S. Baric, Raoul J. de Groot, Christian Drosten, Anastasia A. Gulyaeva, Bart L. Haagmans, Chris Lauber, Andrey M. Leontovich, Benjamin W. Neuman, Dmitry Penzar, Stanley Perlman, Leo L. M. Poon, Dmitry V. Samborskiy, Igor A. Sidorov, Isabel Sola, John Ziebuhr, and Coronaviridae Study Group of the International Committee on Taxonomy of Viruses. The species severe acute respiratory syndrome-related coronavirus: classifying 2019-ncov and naming it sars-cov-2. *Nature Microbiology*, 5(4):536–544, Apr 2020.

[9] Hugo Heyman and Love Brandefelt. A comparison of performance & implementation complexity of multithreaded applications in rust, java and c++, 2020.

[10] Paul B. Jensen. *Cross-Platform Desktop Applications: Using Node, Electron, and NW.js*.

[11] Steve Kinney. *Electron in Action*.

[12] S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[13] Elias Müller. Web technologies on the desktop: an early look at flutter, 2021.

[14] Lucas Fernandes Nogueira. v0.3.0 - tauri alpha. `https://github.com/tauri-apps/tauri/releases/tag/v0.3.0`, Dec 2019. [Accessed 13-Aug-2022].

[15] E.R. Pernice et al. Application Development in the Face of Evolving Web Technologies at the National Ignition Facility. In *Proc. ICALEPCS'19*, number 17 in International Conference on Accelerator and Large Experimental Physics Control Systems, pages 1052–1056. JACoW Publishing, Geneva, Switzerland, 08 2020. https://doi.org/10.18429/JACoW-ICALEPCS2019-WEMPR006.

[16] The Chromium Projects. Blink (rendering engine). `https://www.chromium.org/blink/`. [Accessed 13-Aug-2022].

[17] Kevin Sawicki. Atom shell is now electron. `https://www.electronjs.org/blog/electron`, Apr 2015. [Accessed 26-May-2022].

[18] Gian Luca Scoccia and Marco Autili. Web frameworks for desktop apps: An exploratory study. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and*

Measurement (ESEM), ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Kristijan Stefanoski, Aleksandar Karadimche, and Ile Dimitrievski. Performance comparison of c++ and javascript (node. js–v8 engine). *Research Gate*, 2019.

[20] `https://github.com/tauri-apps`. Tauri. `https://github.com/tauri-apps/tauri`. [Accessed 13-Aug-2022.

[21] `https://www.electronjs.org/`. Electron. `https://github.com/electron/electron`. [Accessed 30-Jul-2022.

[22] `https://www.electronjs.org/`. Electron documentation. `https://www.electronjs.org/docs/latest`. [Accessed 13-Aug-2022].

[23] WebView. Webview. `https://github.com/webview/webview`. [Accessed 16-Aug-2022.

[24] David Wood. *Polymorphisation: Improving Rust compilation times through intelligent monomorphisation*. PhD thesis, MS thesis, School of Computing Science, University of Glasgow, Glasgow, Scotland, 2020.

[25] Cheng Zhao. Release v0.1.0: Update node: use node's implementation of set immediate. `https://github.com/electron/electron/releases/tag/v0.1.0`, Jul 2013. [Accessed 26-May-2022].