Aki Huhta

Multi-platform data processing engine

School of Technology and Innovations
Master's thesis in Automation and Computer Science
Energy and Information technology

Vaasa 2021

UNIVERSITY OF VAASA
School of Technology and Innovations
Author: Aki Huhta
Title of the Thesis: Multi-platform data processing engine
Degree: Master of Science (Tech.)
Programme: Automation and Computer Science
Supervisor: Jouni Lampinen
Instructor: Lassi Niemistö
Year of completing the thesis: 2021 **Pages:** 69

ABSTRACT:
Modern software often must run on multiple different platforms, devices, CPU architectures and software stacks. To simplify development of software and to minimize implementation mistakes, it is often desired to reuse a single implementation in multiple platforms instead of developing and maintaining another implementation of the software.

The objective of this thesis is to find a technology for a medium sized software company. The technology should allow running the same code on three platforms, which are web browsers, servers, and edge devices.

This thesis consists of two parts. The first part describes multi-platform computing, and its history, its common problems and the runtime environments related to the case company's problem. The second part defines the requirements for the chosen technology, selects a set of technologies to review in detail, reviews the chosen technologies and implements and benchmarks a proof-of-concept.

The study resulted in a recommendation of a technology to the case company. The study also identified the constraints and problems that the technology has. Some recommendations for future development of multi-platform software were given.

In the study it became clear that the greatest constraints for solving the problem were the web browser environment and edge devices. The choice in technologies for the web is not as wide as in the case of servers and edge devices. Also, the limited resources of edge devices were an issue. The study found out that in multi-platform software a constraint on one of the platforms applies to all the platforms used. Other general observations were also made.

KEYWORDS: Multi-platform software, cross-platform programming, edge computing, V8 embedded

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen yksikkö
Tekijä: Aki Huhta
Tutkielman nimi: Multi-platform data processing engine
Tutkinto: Diplomi-insinööri
Oppiaine: Automaatio- ja tietotekniikka
Työn valvoja: Jouni Lampinen
Työn ohjaaja: Lassi Niemistö
Valmistumisvuosi: 2021 **Sivumäärä:** 69

TIIVISTELMÄ:
Moderneja ohjelmistoja usein käytetään monella eri alustalla, laitteella, prosessoriarkkitehtuurilla ja ohjelmistopaketilla. Ohjelmistokehityksen yksinkertaistamiseksi ja virheiden vähentämiseksi toteutuksissa yleensä toivotaan, että yhtä toteutusta voitaisiin käyttää monella eri alustalla sen sijaan, että kehitettäisiin ja ylläpidettäisiin uutta toteutusta ohjelmistosta.

Tämän opinnäytetyön tavoitteena on valita teknologia keskisuurelle ohjelmistoyritykselle. Kyseisen teknologian tulisi mahdollistaa saman koodin ajamisen kolmella alustalla, jotka ovat webselaimet, palvelimet ja edge-laitteet.

Opinnäytetyö koostuu kahdesta osasta. Ensimmäinen osa kuvailee alustariippumatonta tietojenkäsittelyä, sen historiaa, sen yleisiä ongelmia ja toimeksiantajayrityksen ongelmaan liittyvät suoritusympäristöt. Toinen osa määrittelee valittavalle teknologialle asetettavat vaatimukset, valitsee joukon teknologioita tarkempaan arviointiin, arvioi valittuja teknologioita ja toteuttaa prototyypin sekä mittaa sen suorituskykyä.

Tutkimuksen tuloksena annettiin suositus teknologiasta toimeksiantajayritykselle. Tutkimus myös tunnisti rajoitteet ja ongelmat, joita kyseisellä teknologialla on. Tulokset sisälsivät myös joitakin suosituksia alustariippumattoman ohjelmiston kehitykseen tulevaisuudessa.

Tutkimuksessa tuli selväksi, että suurimmat rajoitteet ongelman ratkaisemiseksi olivat webselainympäristö sekä edge-laitteet. Vaihtoehtoja teknologioille webselaimia varten ei ole yhtä paljon kuin palvelimille ja edge-laitteille. Edge-laitteiden rajalliset resurssit olivat myös ongelma. Tutkimuksessa selvisi, että alustariippumattomassa ohjelmistossa yhden alustan rajoite pätee kaikkiin käytettyihin alustoihin. Myös muita yleisiä havaintoja tehtiin.

AVAINSANAT: Alustariippumaton ohjelmisto, monialustaohjelmointi, reunalaskenta, V8 embedded

Contents

1 Introduction

Multi-platform computing is common, as software is run on various devices that might have different platforms, CPU architecture, software stack and hardware. For example, it is very common to have an application running on mobile devices and web browsers. Another example is edge computing, where data might be gathered from many different devices and the edge module must run on all those data gathering devices. Multi-platform software is not a new phenomenon. Software has always targeted multiple different targets due to the wide spectrum of hardware in use.

Multi-platform or cross-platform software allows using a single implementation (or code base) on all the different platforms and devices the software is used on. This has multiple benefits; simplicity of software development, organizational simplicity (one team versus multiple) and higher surety that the software has the same functionality on all targets.

1.1 Background and motivation

This thesis topic comes as an assignment from a medium-sized Finnish software company that is the employer of the author. The company has a data processing engine, that currently has three different implementations. One implementation is in JavaScript and is used in the web browser, second implementation is in Java and runs on a server and the third implementation is in Lua and runs on edge devices. The JavaScript and Java implementations are in use currently, while the Lua implementation is not in use.

The company would like to have the data processing engine on edge devices due to customer requests. Currently two implementations are in use and they have different code bases in different programming languages. Maintaining and developing two code bases simultaneously is time consuming and difficult. The two implementations have differences in the functionality they support and there is no guarantee that the shared functionality behaves similarly in the two implementations. Adding a third implementation in a third programming language to use would worsen these issues. Every new

functionality would possibly be implemented three times and the different constraints of the three technologies would lead to the supported features for each platform diverging from one another. Thus, a solution for this issue is required.

1.2 Research questions and objectives

The objective of the thesis is to recommend the company a technology to solve their issue regarding the data processing engine on edge devices. The chosen technology should allow using a single implementation of the core functionality of the data processing engine in all three different platforms (web browser, server, and edge devices).

The data processing engine implements logic blocks that perform a single task. Each platform would still likely have platform-specific blocks, such as graphical user interface manipulation in JavaScript for the web browser, but each block should be implemented only once. A core set of blocks implemented with one technology is needed. This would make the development process easier and simpler, and there would be certainty that the data processing engine behaves similarly on all platforms.

The research questions are as follows:

- What technology allows using a single implementation on all three platforms?
- Can an existing implementation be taken into use on all three platforms, or is a new implementation required?
- What are the limitations of the chosen technology?
- What is the performance of the chosen technology?

The thesis' research will be targeted to solve the case company's problem, but the results can be used in general for similar issues. A recommendation of technology for further evaluation is given. A proof-of-concept is developed and tested, but further analysis, complete implementation and testing is left for further research.

1.3 Structure of the thesis

The thesis will consist mainly of two parts. The first part presents multi-platform computing, its history, use cases and common problems as a literature review. This part will also describe the target platforms relevant to the case company's problem using relevant research and documentation. The case company's software implementations will also be presented.

The second part consists of the author's contribution. This part defines the requirements for the technology choice based on the opinions of experts at the case company, the company's internal documentation and the constraints of the platforms being targeted. In this part a set of technologies to review is selected and then reviewed. A proof-of-concept of the most promising technology is implemented, and its performance is tested.

The results of the technology review, proof-of-concept implementation and testing are presented after the second part. Finally, the author draws conclusions regarding the technology choice and gives recommendations for the company and for anyone encountering similar problems.

2 Current situation and the platforms

In this chapter the current situation of the case company's data processing engine will be described on a high level. Also, the three platforms, the three programming languages used in the software's current implementation, and some related technologies are presented to give enough background for the rest of the thesis. The differences of the platforms and their requirements and the variety in programming languages used will become clear.

2.1 Data processing engine

The data processing engine that is the focus of this thesis implements a set of logic blocks, that are combined by users into flows to be executed on the selected platform. The logic blocks implement operations such as bit manipulation, logical operators, arithmetic operations, and signal processing. As the operations are primitive, the idea of a unified codebase is sensible.

The data processing engine is implemented in three different programming languages for three different platforms. One implementation is done in Java and is run on a server in the cloud. The second implementation is done in JavaScript and is run on a web browser on the user's machine when viewing the web application. The third implementation, that is not currently in use, is done in Lua and is supposed to be used on edge devices.

Currently the Java and JavaScript implementations have almost the same blocks, with JavaScript implementation having the most blocks and the Java implementation missing some blocks. The Lua implementation is missing some blocks that the other two implementations have, but on the other hand it has some blocks that implement functionality specific to embedded use cases, such as signal edge detection. It has been identified that all the implementations have a core set of blocks, that could be shared between all of them.

Because all the implementations use a different programming language, the code, its style, organization, and architecture are different for each implementation. For example, the Java and JavaScript implementations have implemented one block per source file whereas the Lua implementation has all its blocks in a single source file. The JavaScript implementation has organized the blocks into directories based on their categorization (logic, data, calculation, miscellaneous), but the Java implementation has them all in a single directory. The implementations are stored all in different Git repositories.

2.2 Server

In a client-server computing clients and servers form a system that allows distributed computing, analysis, and presentation. A client can be a process interacting with user that provides the user interface (UI) used by the user for data retrieval, analysis, and presentation. A server provides the client with services, which are defined by the business goals. The service could be print server or file server requiring minimal server-based computation, or database server or image processing requiring intensive computations. A server responds to queries and commands from a client. Generally, the server does not initiate the communication with the client. (Sinha, 1992)

In the case company's system, the server uses the data processing engine to process the data sent by the client using the steps defined by the client when it is not desirable or feasible to process in the client. The client can be the web browser used by a user or it can be another system that requests and uses data or provides the data.

2.2.1 Java

Java is an object-oriented, interpreted, and portable programming language with garbage collection for automatic memory management. The Java platform supports multithreading, dynamic loading of code modules and has built-in tamper-protection. (Gosling & McGilton, 1996)

Java originated as a research project where the aim was to develop advanced software for various network devices and embedded systems. The goal was a small, reliable, distributed, portable and real-time operating system. Originally, the code was developed in C++, but due to difficulties faced in development, an entire new programming language was created. (Gosling & McGilton, 1996)

Java was designed to answer to the needs of development in heterogenous and network-wide distributed environments. To answer these needs, the main challenges are to deliver securely applications that consume minimal resources and can run on any hardware or software platform while having the option of being dynamically extended. For these reasons, Java was designed to be architecture neutral, portable, and dynamically adaptable. (Gosling & McGilton, 1996)

Java as a programming language has been designed to be simple and object-oriented from the ground up. Java also has a wide variety of extendable libraries. Portability is provided by Java Virtual Machine (JVM), on which the bytecode generated by the Java compiler is run on. The JVM is based on the POSIX interface specification and the implementation of JVM on new architectures is possible and straightforward if the target platform meets a few basic requirements such as multithreading. The bytecode is architecture neutral intermediate format. Portability is also helped by having the same data types and arithmetic operator behavior across platforms. The compiler has compile-time static type checking, but the language and runtime are dynamic in the linking stage – classes are linked only if needed and new modules can be linked on demand. Java's memory management is simple – it has no programmed-defined pointer data types, no pointer arithmetic and objects are created with simply a *new* operator and garbage collection automatically frees the memory. (Gosling & McGilton, 1996)

Java also can interoperate with other programming languages which is important for this thesis as any solution selected should also integrate to the Java codebase on the server. Java Native Interface and Java Scripting API could help in this regard.

2.2.2 Java Native Interface

Java Native Interface (JNI) is a native programming interface that allows Java code running inside a Java Virtual Machine to interoperate with code written in other programming languages. The JNI is required when the standard Java class library does not support platform-specific features, when a library written in another language is used with Java code, or when time-critical code is implemented in a lower-level language such as assembly. JNI can be used to create, inspect, and update Java objects, to call Java methods, to catch and throw exceptions, to load classes and obtain their information and to perform runtime type checking. (Java Native Interface Specification, n.d.)

2.2.3 Java Scripting API

The Java Scripting API is a framework for using script engines from Java code and it is independent of any scripting language (Java Scripting Programmer's Guide, n.d.). Originally, the Java Development Kit (JDK) came with a script engine for JavaScript called Nashorn builtin, but it has been deprecated (see JEP 372: Remove the Nashorn JavaScript Engine, n.d.). The GraalVM virtual machine from Oracle can be used as a replacement (see Migration Guide from Nashorn to GraalVM JavaScript, n.d. and Oracle/graaljs, n.d.).

2.2.4 Container-technology

Containers are a technology for separating an application from the operating system and the physical infrastructure used for networking. A container is instantiated in the kernel and it virtualizes an instance of an application. They are used for example to sandbox applications or by Software-as-a-Service (SaaS) providers to isolate the applications and data of different customers. (Hogg, 2014)

Hogg (2014) explains that Linux Container (LXC) isolates the CPU, memory, file, I/O, and network resources with control groups (cgroups), and it also uses namespaces to isolate the application from the operating system. According to Hogg, LXC separates the process trees, user IDs, network and file access. The benefit of LXC according to Hogg is that it allows virtualizing a single application instead of virtualizing the entire operating system with a virtual machine. Hogg mentions that the popular container technology, Docker

which adds image management and deployment tools, was originally built on top of LXC. Later though Docker has developed its own technology for containers.

The case company uses container technology with the server-side codebase; the application is run in a container as the server environment is in the cloud. The container technology is not expected to pose any problems with the selection of the technology in this thesis, but it is kept in mind when seeking technologies.

2.3 Web browser

A web browser retrieves information from the web (WWW) and displays it on the device. Information is transferred using Hypertext Transfer Protocol (HTTP), that defines how data consisting of text, images and videos is transmitted. After the data is fetched, the browser uses a rendering engine to translate Hypertext Markup Language (HTML) to text and images. Browsers support hyperlinks allowing links to other resources on the Web, each of which has a unique Uniform Resource Locator (URL) also known as an address. The address tells the browser which server to request data from. (Mozilla, n.d.)

In addition to HTML, web browsers also understand Cascading Style Sheets (CSS), which is used to define how the components specified by the HTML look like. To bring functionality and interactivity by programming, web browsers support JavaScript and WebAssembly.

Web browsers usually follow web standards, that define how web technologies should work. World Wide Web Consortium (W3C) standardizes HTML and CSS, whereas JavaScript is standardized by Ecma International.

Commonly used web browsers include Mozilla Firefox, Google Chrome, Microsoft Edge and Apple Safari.

2.3.1 JavaScript

JavaScript was created by Brendan Eich at Netscape. Originally Netscape was looking for a language inside a browser that could be used to automate parts of a web page or make it more dynamic. The language was called first LiveScript but was later renamed to JavaScript. JavaScript was the answer to the need of doing things that HTML was not able to express – make things move, respond to user input, change colors, ask input with a dialog box. A draft standard of JavaScript was submitted to European Computer Manufacturers' Association (ECMA), a communication standards body, which then adopted the standard. The standard now calls the language ECMAScript. (Andreessen, 1998)

As a programming language, JavaScript is lightweight, interpreted, and object-oriented and it has first-class functions. The object-orientation is prototype-based, and the language also supports imperative and functional programming. The type system is dynamic. In JavaScript, objects are created programmatically by adding methods and properties to empty objects at runtime. This is different from the class definitions in languages like C++ and Java. The syntax of JavaScript is similar to Java and C++. (MDN Web Docs, 2021a.)

2.3.2 WebAssembly

Wagner (2017) writes that Alon Zakai, a Mozilla employee, had the idea of converting a game written in C++ to JavaScript that can be run on the Web. Wagner continues that this Zakai's endeavor ended up becoming a software called Emscripten. At the time, Emscripten targeted a subset of JavaScript, asm.js, according to Wagner. He explains that the standardization efforts of this approach of targeting the Web ended up with the birth of WebAssembly.

Originally web developers had two choices; they could use HTML, CSS and JavaScript to create applications running in the web browser or they could create browser plugins that the users would download and install. With these two choices problems were often encountered if certain kind of applications were developed. The first approach had

mediocre performance with compute-heavy tasks. The plugins required users to download and install possibly malicious code, and the plugin were browser specific. WebAssembly is a technology to solve this problem. (Wagner, 2017)

WebAssembly (Wasm) is a binary instruction format for a virtual machine and is designed to be a portable compilation target. WebAssembly enables deployment on the web. WebAssembly aims to execute at native speed while providing a memory-safe, sandboxed execution environment. WebAssembly modules can also access browser functionality through the same APIs as JavaScript. WebAssembly is not meant only for the web, but also supports usage in other environments. (WebAssembly.org, n.d.-a)

The high-level goals of WebAssembly are to define a potable, size-efficient and load-time-efficient binary format that can be used as a compilation target and taking advantage of common hardware capabilities to achieve high performance. It is designed to execute and integrate with the existing Web platform. The use cases are for example better execution for languages such as C/C++ which are cross-compiled to the web, image and video editing, games, encryption and simulation in the browser. Outside the browser, use cases can include game distribution, server-side computation of untrusted code, server-side applications, and symmetric computations across many nodes. (WebAssembly.org, n.d.-d, WebAssembly.org, n.d.-e)

WebAssembly System Interface (WASI) is an API that provides access to operating-system-like features such as filesystems, Berkeley sockets, clocks, and random numbers. WASI is designed to be independent of browsers. Currently C/C++ and Rust toolchains can take advantage of WASI. WASI can used on the Wasmtime WebAssembly runtime or on the browser using a polyfill. (Bytecodealliance/wasmtime, n.d.)

2.4 Edge device

Edge computing is about enabling technologies that allow computation to happen near the data sources. An edge device is any computing or networking resource located in-between data sources and cloud-based datacenters. For example, a smartphone can be

an edge device between body sensors and the cloud. In edge computing, the edge devices both consume and produce data. They request services and information from the cloud and handle computing tasks such as processing, storage, caching and load balancing. (Shi & Dustdar, 2016)

2.4.1 Lua

Lua is a general-purpose embedded programming language that is designed for supporting procedural programming with data description capabilities. As it is an embedded language, it does not work without a host. Lua is a library of C functions that are linked to the host application. The host can invoke functions from the library to execute a piece of code in Lua, to write and read Lua variables and to register C functions to be called by Lua code. As a programming language Lua has first-class functions, object-orientation, dynamc typing. Lua uses Pascal-like syntax. (Ierusalimschy, Figueiredo, & Filho, 1996.)

3 Multi-platform computing

Multi-platform or cross-platform computing refers to developing software for, or running software on, multiple different types of hardware platforms. An example of multi-platform software could be the web browser, which commonly runs on desktops, laptops, and mobile devices and on different operating systems. (PC Magazine, 2021)

In this context, platform can mean different processor architectures, operating systems, or hardware configurations. Each processor architecture can demand compiling the software to a different instruction set, each operating system might have different demands for the software and their own APIs for accessing the hardware. The hardware on different devices can vary; combinations of CPU, GPU, RAM, storage devices and so on are plenty.

The case company is targeting many platforms, both hardware and software. The browser implementation targets the common web browsers that in turn target many operating systems. The server implementation targets the Linux operating system, a container platform and the x86 architecture. The edge device implementation targets a Yocto-based custom Linux distribution and the ARM architecture. This shows that the real-world cases of multi-platform computing are often complex.

Cusumano & Yoffie (1999) note that there are two ways of creating cross-platform products: develop separate platform-specific versions of the product or develop the bulk of the product in generic, cross-platform code, with little or no code tailored to different platforms. In this thesis the interest is in the latter method and it fits into the definition of multi-platform computing given in this section.

PC Magazine's (2021) definition for cross-platform software gives two main methods of developing cross-platform software: compile an executable program to the operating environment of each target computer or use an intermediate language and compile only once.

3.1 Multi-platform computing in different contexts

In some sense, multi-platform computing is nearly as old as computing itself. There has always been a wide variety of computing hardware and the question of reusing software on different machines has appeared often and been solved numerous times in different contexts. In this sub-chapter an overview of various points and notable developments in the history of multi-platform computing to this date is presented. It will become clear that many approaches have been used to target the many devices in existence.

Compilers were the first technology to enable multi-platform computing. Aho et al. (2007, pp. 1-2) define compilers as programs, that read a program written in one language (source language) and translate it to an equivalent program in another language (target). Aho et al. (2007, pp. 2) explain that if the target program is an executable machine-language program, it can then be executed by the user to process data. If the compiler supports targeting different architectures, then it also enables multi-platform computing; the same source code can be compiled to multiple targets. For example, the GCC (GNU C Compiler) allows targeting many architectures such as the common x86 and ARM architectures (Free Software Foundation, n.d.).

In addition to compilation, there are other methods to process programming languages. Interpreters are software, that instead of giving an output in the target language, execute the operations specified by the source code. Compilation and interpretation can be combined: first the source program is compiled into a bytecode, an intermediate form, and this bytecode is then interpreted by a virtual machine. The Java programming language is an example of this. (Aho et al. 2007, pp. 2-3)

Virtual machines are software, that implement a computer and operating system dependent machine architecture. A compiler can then translate a program to target the virtual machine's instruction. A compiled program targeting the virtual machine is independent of the platforms that run the virtual machine. Additionally, the virtual machine

can use just-in-time technology to translate the virtual machine's instructions to the machine's native instructions. (Bishop & Horspool, 2006)

Java and the .NET Common Language Runtime are examples of widely used virtual machines that execute compiled bytecode. As a virtual machine is implemented on many platforms, such is the case with Java, then the compiled programs are effectively platform independent. (Bishop & Horspool, 2006)

The procedures of using compiled and interpreted programming languages are shown in Figure 1. Interpreted languages have an extra step where the interpreted languages have to be executed on a virtual machine whereas compiled languages allow directly executing the compiled program.

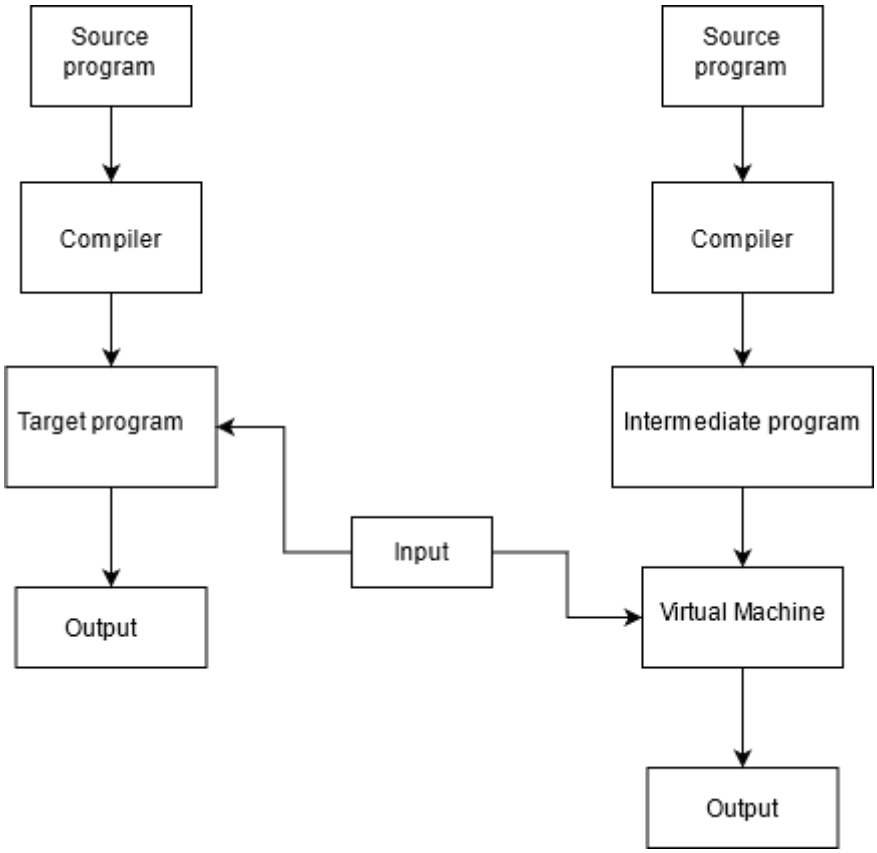

Figure 1 The processes of using compiled and interpreted programming languages.

The original promise of Java's developer, Sun, was "write once, run everywhere". This slogan originates from the fact that developers do not write Java code to run on the APIs of a specific operating system but instead target Java virtual machine. (Cusumano & Yoffie, 1999)

Despite it being possible to compile to different architectures or target a virtual machine, a common issue in desktop software persists – developing Graphical User Interfaces (GUI) for different operating systems. Operating systems have different graphics APIs and by using those APIs, the developer would have to develop a separate GUI for each platform. This is commonly solved with cross-platform GUI toolkits such as Qt.

Web browsers were the next major point in the history of multi-platform computing. Taivalsaari et al. (2008) predicted that the web browser will in essence be the de facto operating system. Taivalsaari et al. (2008) explained that in their belief vast majority of future software applications will target the Web and browsers instead of the conventional target platforms such as specific operating systems and CPU architectures. Now in 2021 it is quite clear that their belief was correct.

One of the reasons for this explosive increase in web applications might be the ease of targeting multiple platforms. If web browsers are implemented for example on Windows, Linux and macOS computers, and on mobile devices and their different operating systems, then an application developer can target the web browser instead of writing applications specific to these platforms. Taivalsaari et al. (2008) noted that it was at the time becoming easier to develop desktop-style web sites or web applications although they also identified many issues with the web browser as an application platform. Nevertheless, web applications became popular. There was a tradeoff between portability and functionality when choosing to develop web applications or conventional desktop applications. Wagner (2017) explains the draw of the web as a platform: what if you could share a Computer Aided Design (CAD) model with a colleague, asking them to

modify it without any need to instruct them to install special software or without worrying if the software is compatible with their environment?

With the popularity of web applications, there were more and more supply of web developers. Many companies had JavaScript, HTML and CSS experts as just about every business needed a web page or web application. The question of using web technologies on the desktop without the browser was natural. JavaScript, HTML and CSS were supported by web browsers on multiple operating systems already. HTML and CSS were found to be great for developing cross-platform user interfaces. Electron was a technology that answered this need.

Electron is a tool for building cross-platform desktop applications with the web technologies: JavaScript, HTML and CSS. It is based on Chromium and Node.js. Electron allows building for and running applications on Linux, Windows, and Mac. Popular applications built on Electron include Visual Studio Code, Facebook Messenger, and Microsoft Teams. (Electronjs.org, n.d.)

As Electron allowed bringing code from the web browser environment to the desktop, WebAssembly on the other hand allows bringing code from other platforms to the web browser. Now that programming languages such as C/C++ and Rust can be compiled to WebAssembly, it is possible to share code between the web browser environment and other platforms.

Smartphones were the next major development after web browsers. Smartphones, like the traditional desktops and laptops, also have often the need for multi-platform support. Currently the smartphone market is dominated by Android and iOS operating systems; according to Gartner (2017) in the first quarter of 2017 Android's share of sales was 86.1% and iOS's 13.7%. Android and iOS do not have a compatible API. Instead of developing separate applications for both major smartphone operating systems, many developers choose to use cross-platforms tools such as React Native, Cordova and

Xamarin as seen from Stack Overflow Developer Survey 2019 (2019a). These cross-platform toolkits allow using the same code on both major operating systems once again solving the problem of targeting multiple platforms.

To summarize, the problem of targeting multiple platforms, whether they are hardware of software platforms, is solved software that handles the difficult multi-platform targeting. The tools used include compilers, interpreters, virtual machines, cross-platform GUI toolkits, cross-platform SDKs.

3.2 Common problems of multi-platform computing

The variety of hardware in devices nowadays is broad. All computing devices have the same basic components, but due to various use cases of the devices they all have different constraints placed on the hardware. A mobile devices processor cannot have as high of a performance as the processor of a desktop because the mobile device has more constraints placed on it. Power supply, cooling and available physical space for the CPU are all more constrained on the mobile device versus the desktop. Edge devices often have even less computing power than mobile devices; they might be mass produced leading into the need for cheap components. Edge devices can be used in for example factories, where they might require hardening against demanding environment.

In addition to hardware, the developer of multi-platform software must consider the available technologies. Finding a technology, programming language or framework that can target all desired devices can be difficult. Even if the technology can target all the devices, then there is the question whether it behaves similarly on all the devices in question.

When developing the multi-platform software, there is another question to consider. Should the software be developed to behave the same on all devices with no platform-specific code or should the software have some platform-specific code? Having some platform-specific code would certainly help take advantage of the strengths of each

platform but it would also lead into losing some of the benefits of sharing the same code on all platforms such as code reuse, simplicity and guarantee of similar behavior across devices. Cusumano & Yoffie (1999) claim it is almost always necessary to have some code tailored to different platforms specifically.

Having even a small amount of code tailored to a specific platform causes logistical issues. Different code bases and development teams must be synchronized. The organization must keep track of all the variations in the code and test all versions and changes. On the other hand, minimizing platform-specific code has its own issues. It would require the developers to not use any interfaces or programming constructs specific to an operating system or hardware platform. Developers should instead use simple or low-level programming constructs and interfaces common to all the platforms. Usually many platform-specific interfaces and programming “tricks” enable developers to write code that is often faster or more efficient than the code that uses the lowest-common-denominator interface. Therefore, multi-platform products can be slower to develop or even have weaker functional performance. (Cusumano & Yoffie, 1999)

4 Requirements and technology selection

During discussions with experts that work on the software in question, several requirements for the technology were identified. The case company's internal documentation also supported requirements gathering. These requirements were used to narrow down the set of technology choices to review in closer detail and to make the decision for choosing the recommended technology. Each requirement added more constraints on the technology selection leading into a small set of possible choices.

The technologies were sought by using both Google and Google Scholar searches and links in WebAssembly.org documentation. Search terms used in search included *WebAssembly*, *WebAssembly Compiler*, *JavaScript transpiler*, *JavaScript Compiler*, *WebAssembly Runtime*, *JavaScript to C, JavaScript engine, JavaScript embedded, JavaScript on edge, Javascript multi-platform, Javascript cross-platform*.

4.1 Requirements specification

The requirements for the technology are presented in Table 1. All of these should be fulfilled by the chosen technology.

Table 1 Requirements for the technology.

R1	The same code can be used on all three platforms (web browser, server, edge device)
R2	Code can target JavaScript / WebAssembly and run in the web browser
R3	Code can be run on a Linux-based server in the cloud
R4	Code can be run in a container
R5	Code can be run on a specific Linux-based edge device
R6	Code can be run on x86 and ARM architectures
R7	Technology has high performance
R8	The runtime fits into roughly 100 MB
R9	Well-established technology as opposed to a niche technology
R10	The technology can be integrated into the existing code base

R11	The license of the technology must allow mass-production of devices and using the technology as a part of a product

R1 is the fundamental goal of the study, and R2-R6 give additional details to that requirement. In the web browser, it is possible to execute only JavaScript and WebAssembly (R2). The server platform in this case is Linux-based (R3) and the software is run in a container (R4). The servers' CPUs likely use x86 architecture (R6). The edge device is Linux-based (R5), uses an ARM based CPU (R6) and has limited hardware capabilities (R7, R8). Finally, a well-established technology is desired instead of unmaintained or hobby projects (R9) and the technology should also integrate with the existing code bases on all three platforms (R10). The licensing model of the technology should allow the case company to use it as a part of a commercial product (R11). Specifically, GPLv3 is known to be problematic and unfit for this purpose.

High performance (R7) and small memory footprint (R8) are required due to the hardware limitations of the edge devices the case company is targeting. The edge device of the case company has ARM Cortex-A5 536 MHz CPU, 256MB of DDR2 RAM and 256MB of NAND Flash storage. The CPU in question supports ARMv7-A architecture, which is a 32-bit architecture.

4.2 Selection of technologies to further review

R2 states that the technology should allow running code in the web browser. Because the commonly used web browsers only support running code written in JavaScript or compiled to WebAssembly, this requirement ends up limiting the choices the most.

The first option is to use JavaScript on all platforms. This would require using some JavaScript engine on server and edge device or transforming JavaScript to some other language for the server and edge device. Regarding the latter option, no tools that allow translating JavaScript code to another language such as C, or tools that transpile code to JavaScript were found to fit the requirements.

JavaScript could be run on the server and edge device using a general-purpose JavaScript engine such as the V8 JavaScript engine (see V8.dev, n.d.-b.). Also, other JavaScript engines, that are aimed for embedded devices, such as DukTape, XS, mJS and JerryScript exist (see Duktape, n.d., Soquet, P., 2017, Cesanta/mjs, n.d. and Jerryscript.net, n.d.). According to a benchmark performed by Bellard (n.d.) the performance of these JavaScript engines aimed at embedded devices is significantly inferior to the performance of V8 (when using JIT technology) although the executable size of V8 is larger.

The second option would be to use a programming language that can be compiled to WebAssembly. The browser side would use the WebAssembly and on the server and edge either the language is compiled to the native architecture if possible, or it is compiled to WebAssembly and executed using some WebAssembly runtime. It could also be possible to compile a Lua virtual machine to WebAssembly and use that in the browser platform, but no such serious project was found.

Webassembly.org (n.d.-c) lists programming languages that allow the compilation of WebAssembly modules: C/C++, Rust, AssemblyScript, C#, F#, Go, Kotlin, Swift, D, Pascal, Zig. Of these C/C++, Rust and AssemblyScript fit into the requirements R7 and R9, and their support for WebAssembly was deemed the most mature based on their documentation.

MDN Web Docs (2021b) also lists the following four as the main options for targeting WebAssembly :

- Compiling C/C++ with Emscripten to WebAssembly
- Writing WebAssembly directly
- Compiling Rust to Webassembly
- Compiling AssemblyScript to WebAssembly

Out of these the option of writing WebAssembly directly is discarded as it would make software development too difficult and slow.

If AssemblyScript is compiled to WebAssembly, then WebAssembly runtime is required to run the code on the server and edge device as AssemblyScript does not compile to native code. The only WebAssembly runtime that supports ARMv7 architecture was found to be WebAssembly Micro Runtime (see Bytecodealliance/wasm-micro-runtime, n.d. and Bytecode Alliance, n.d.). Other WebAssembly runtimes include wasmtime and Wasmer but they lack ARMv7 support.

In Table 2 the licenses of the different technologies mentioned in this section are listed. The various programming languages were left out of the listing as programming languages and compilers generally have permissive licenses such as MIT or Apache 2.0 license. Only the license of XS might be problematic out of these, but XS and the other lightweight JavaScript engines were already excluded due to performance.

Table 2 Licenses of various technologies that were considered.

Technology	**License**
V8	V8's BSD-style license
DukTape	MIT
XS	LGPLv3 and GPLv3
mJS	GPLv2
JerryScript	Apache License 2.0
Emscripten	MIT
AssemblyScript	Apache License 2.0
WebAssembly Micro Runtime	Apache License 2.0

The process of selecting the technologies is further visualized in Figure 2. As can be seen from Figure 2, JavaScript, C/C++ with Emscripten, Rust and AssemblyScript with WebAssembly Micro Runtime fulfilled the requirements and thus they will be compared in more

detail in the following chapter. In the figure the requirement R2 is depicted at the top as it was found to be the most constraining requirement. Thus, the figure does not include any technologies that cannot be used with JavaScript or WebAssembly.

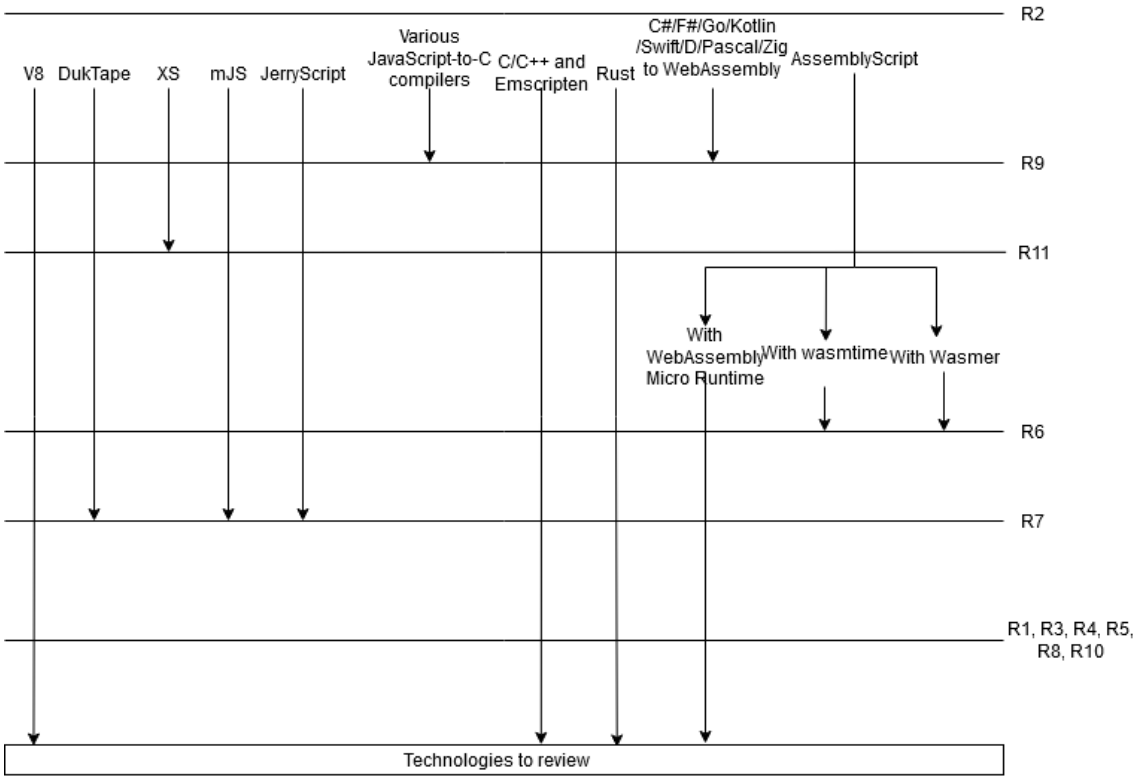

Figure 2 The requirements fulfilled by different technologies.

JavaScript and the V8 engine would be a good choice because the case company's focus is more on the browser and server side. If using JavaScript on the edge device is feasible, then it would be the preferred choice. The relatively large size of the V8 executable when compared to the alternatives should not be an issue as the case company's edge device has enough memory for it. The benefits of better performance make the V8 more desirable option than the others, and V8 is backed by Google versus the other's being open-source projects or maintained by smaller companies.

C/C++ are good options because the case company's existing edge device codebase uses C++ making the integration simple, and C/C++ is known to have high performance and memory efficiency. Furthermore, WebAssembly's initial focus is C/C++ support (WebAssembly.org, n.d.-b). Emscripten is the compiler technology used to target WebAssembly.

Rust is a relatively new programming language, and it aims to have high performance and safety. Rust serves in the review as a contrast to the well-established C/C++. Rust has recently enjoyed some popularity among software developers; Stack Overflow Developer Survey 2019 (2019b) ranked Rust as the "most loved" language. Rust might see more use in the future and developing Rust expertise in the case company could be beneficial.

Finally, AssemblyScript is a TypeScript-like language that compiles to WebAssembly. In the review it serves as the opposite approach of C/C++ or Rust. AssemblyScript would be compiled to WebAssembly and then the WebAssembly would be executed on the server and edge device using a WebAssembly runtime whereas C/C++ and Rust can compile to the instruction set of the target CPU architecture.

5 Technology review

This chapter concerns the technologies that were found to promise to fulfill the requirements defined in the previous chapter and documents the findings of the technology review. The technology review was conducted by reading the official documentation of the chosen technologies and by searching the Web for more information.

These technologies will be described, notable strengths and weaknesses noted by prior literature will be described, and if possible, the following questions will be answered for each technology:

- What is the maturity of WebAssembly or browser platform support?
- Is there a library ecosystem? Is it possible to use existing libraries with WebAssembly?
- Can the technology be integrated to the case company's existing code bases?
- Will the performance especially on low-end hardware be enough?
- Is it possible to access system resources on the server and edge devices?
- Does the code require some browser or WebAssembly specific structures? If so, then does the same code work on server and edge? Or is it possible to write generic code that works on all platforms?

The set of technologies being reviewed is as follows:

- JavaScript and the V8 engine
- C/C++ and Emscripten
- Rust
- AssemblyScript and WebAssembly Micro Runtime

With JavaScript and V8 the focus is on using JavaScript on the edge device with V8 as JavaScript is already used in the browser. Due to Node.js using V8, it is already known V8 can be used on the server. The main question is its performance. Regarding C/C++/Emscripten and Rust, the focus is on WebAssembly side as it is known that these languages will run on the server and edge device. In the case of AssemblyScript and WebAssembly

Micro Runtime the focus is more on integrating the WebAssembly Runtime with the existing codebases as well as the maturity and performance of AssemblyScript.

5.1 JavaScript and V8

V8 is an open-source JavaScript and WebAssembly engine by Google written in C++. It is most known for being used in the Google Chrome web browser and Node.js. It can be used on many architectures including x64 and ARM. It can also run standalone or be embedded into a C++ program. (V8.dev, n.d.-b.)

V8 is likely the most mature platform for executing JavaScript as it is used in Google Chrome and Node.js (see V8.dev, n.d.-b.). Using it would allow also using any existing JavaScript libraries like it is possible to use JavaScript libraries for example when developing a web application backend on top of Node.js.

Figure 3 shows how the V8 engine and JavaScript would fit into the larger system. As the V8 can be embedded into a C++ application and used like a library, integration to the edge device's codebase would be simple enough. On the server Java Scripting API could be used to integrate JavaScript and V8 into the existing codebase. On the server side other engines could also be used if V8 integrations is difficult as the performance is not that constrained as on the edge device. Furthermore, it is a likely option that in the future the whole server side is converted to JavaScript native, i.e., Node.js that is based on the V8 engine.

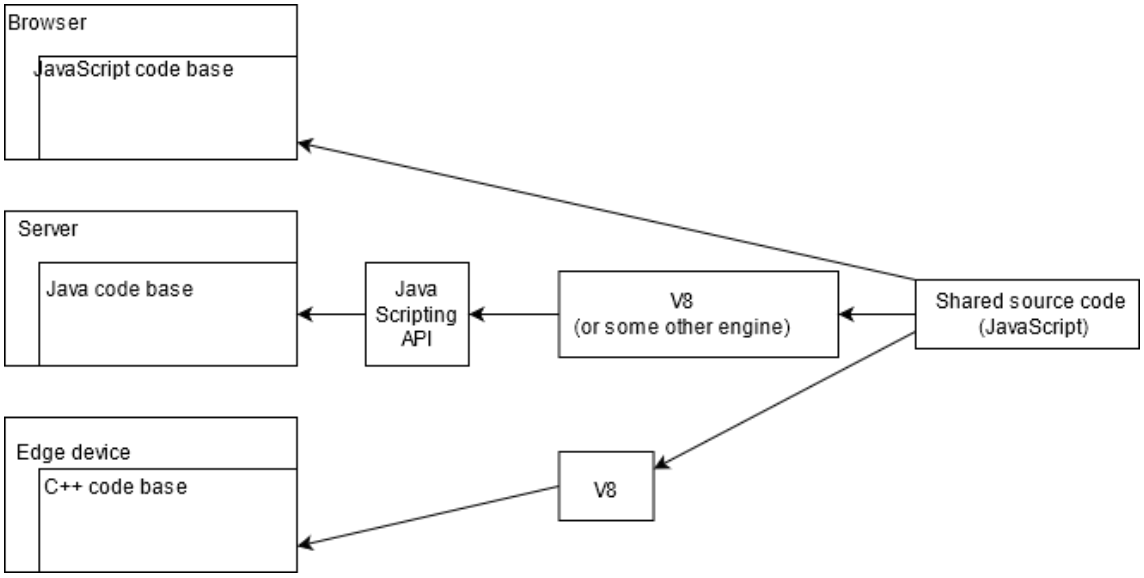

Figure 3 Usage of JavaScript and V8 in the system

Oliveira & Mattos (2020) performed a benchmark on a Raspberry PI3 model B and compared JavaScript and WebAssembly on Node.js and C using the Ostrich Benchmark Suite. Their results are especially interesting for this thesis as their hardware was quite near in performance to the case company's edge device, and Node.js uses V8 engine. Their results show that JavaScript on Node.js had about two to four times the execution time of C, and JavaScript also used more memory and energy than C. This level of performance would likely be enough for the case company.

Finally, accessing system resources from the JavaScript running on V8 and using the same code on all three platforms would be possible. All three platforms would be executed on V8 engine, or a similar engine if browsers other than Google Chrome are used or embedding V8 into Java turns out to be impossible. The programming language used would be the same across all platform making the development simple.

If the performance of V8 on the case company's edge device is found to be adequate, or atleast on a similar level as found by Oliveira & Mattos (2020) then it would be the preferred choice. Using JavaScript and V8 does not seem to have any major weaknesses other than the question of performance on edge devices.

5.2 C/C++ and Emscripten

C is a general-purpose programming language though not very high-level language and is not specialized to any area. Dennis Ritchie originally designed it for the UNIX operating system on the DEC PDP-11. (Kernighan & Ritchie, 1988.)

C++ was originally designed by Bjarne Stroustrup to answer the question of simultaneously directly manipulating hardware and supporting efficient high-level abstraction. In the beginning C++ was a combination of the features of C and Simula programming languages. Now it has grown to a complex and effective tool for a wide range of applications. It started as “C with classes” in 1979. (Stroustrup, 2020. pp. 5-6)

Despite being different programming languages, C and C++ are grouped here together as it does not make a significant difference whether one is used over the other in this case. Their main difference is that C++ has a lot more language features such as classes over C and is more complex in general. They are both known for being suitable for embedded software and high-performance software.

Emscripten is an open-source compiler toolchain for WebAssembly. It enables compiling C and C++ code (or any other language that uses LLVM) into WebAssembly. Almost any portable C/C++ codebase can be compiled using Emscripten into WebAssembly. Emscripten can be used as a drop-in replacement for the common compilers GCC and Clang. it uses Clang and LLVM to compile to WebAssembly while also outputting JavaScript code providing API that supports the WebAssembly code. (Emscripten 2.0.12 documentation. n.d.-a.)

Emscripten is likely the most mature technology when it comes to WebAssembly. Emscripten 2.0.12 documentation (n.d.-a) supports this notion by mentioning that Emscripten has been already used to convert several real-world codebases to WebAssembly and it lists Unreal Engine and Unity as examples. As another example, Google Earth has also been ported to WebAssembly using Emscripten (Chromium Blog, 2019). These examples

show that Emscripten can be used to port large, real and commercial C/C++ projects to WebAssembly.

Emscripten 2.0.12 documentation (n.d.-d) states that Emscripten provides support for standard libraries such as libc, libc++ and SDL, and automatically links them when they are used. It also explains that other libraries can be used if it is possible to build and link them. In other words, the standard library functionality can be used and there is no need to implement it again for the WebAssembly target. Also, third-party libraries can be used if they can be compiled using Emscripten.

Emscripten also provides an API that allows integration with the browser environment, interfacing with the HTML5 API, working with the compiled code from JavaScript etc. (Emscripten 2.0.12 documentation, n.d.-c)

A major benefit to using C/C++ and Emscripten would be the fact that C/C++ can be used on the server and edge device. Figure 4 shows how C/C++ and Emscripten would fit into the larger system. On the server-side C/C++ can be called from Java code using for example Java Native Interface, while on the edge device the codebase is already C++ which would make using C++ trivial and easily allow calling C code.

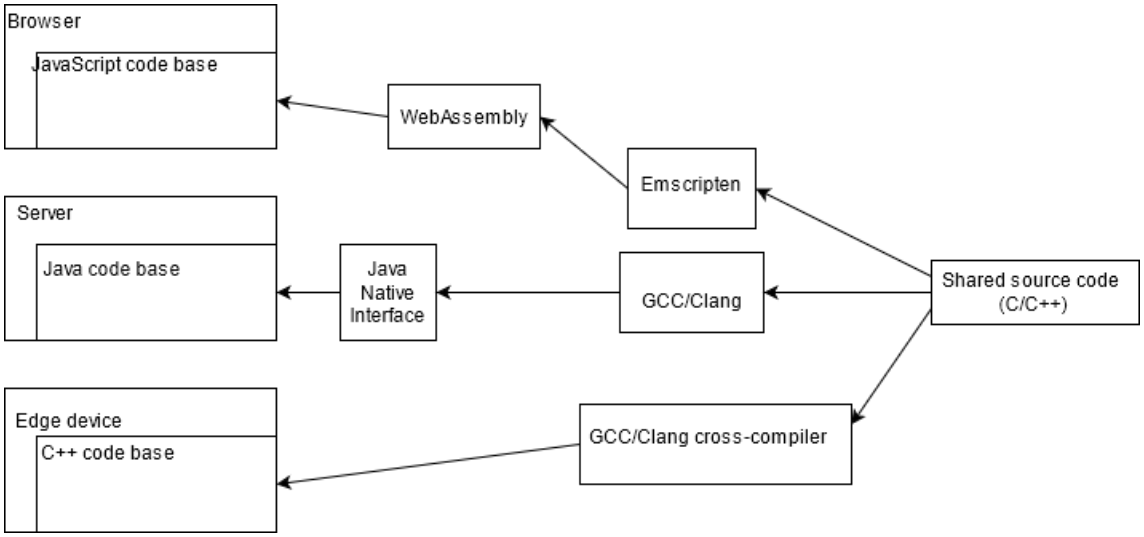

Figure 4 Usage of C/C++ and Emscripten in the system

The performance of C/C++ should pose no problems on the edge device. C and C++ are widely known to be the de facto choice for embedded devices and low-end hardware. Furthermore, the existing codebase on the edge device already uses C++ proving that its performance is enough for the case company's edge device.

Because C/C++ would be compiled to the native architecture on the server and edge device, using system resources is possible. Also, on the browser side Emscripten supports some system functions such as networking (asynchronous operations) and file system functions through a virtual file system (Emscripten 2.0.12 documentation, n.d.-b).

Using the same code on all three platforms should be mostly feasible. As Emscripten 2.0.12 documentation (n.d.-a) explains "*Practically any **portable** C or C++ codebase can be compiled into WebAssembly using Emscripten*". A core use-case of Emscripten is to compile existing applications to WebAssembly. Some platform-specific code is necessary on the three platforms to integrate to the existing codebase, but it should be possible to implement the core functionality of the data processing engine with generic code. Since Emscripten aims to compile large, existing applications, it might even be feasible to implement a larger entity than just the core functionality of the data processing engine.

Instead of merely sharing the core functionality on all platforms, the surrounding logic could also be ported to the web browser.

To summarize the findings: Emscripten is a complete toolchain for compiling C/C++ to WebAssembly. It provides support for standard libraries and glue-code for integration with JavaScript. It has been used by real projects successfully and C/C++ are the initial focus of the WebAssembly project.

5.3 Rust

Rust is a fast and memory-efficient programming language with no runtime or garbage collector. It has an ownership model guaranteeing memory-safety and thread-safety. WebAssembly is one of the domains where the Rust community has decided to improve the programming experience. (Rust Programming Language, n.d.)

Rust began as Graydon Hoare's side project in 2006 and Mozilla got involved in the project in 2009. The original goal was to design a safe, concurrent, and practical systems language. (Frequently Asked Questions · The Rust Programming Language, 2016)

The Rust project appears to be very serious about supporting WebAssembly. The projects homepage mentions WebAssembly on the landing page (see Rust Programming Language, n.d.). Rust has also seen production use with regards to targeting WebAssembly: Horn (n.d.) writes that Dropbox were able to embed a codec written in Rust in a webpage using WebAssembly, according to Pack (2018) Cloudflare compiles Rust to WebAssembly and calls it from serverless functions and Fitzgerald (2018) explains how he and Tom Tromey ported the the performance-sensitive portions of the source-map JavaScript library to Rust and WebAssembly.

Crates [basically libraries in Rust terminology] that avoid things that do not work with WebAssembly tend to be portable to WebAssembly. If the crate supports embedded and the *![no_std]* directive, that is crates that do not rely on the standard library, then it likely supports WebAssembly. Anything that uses system libraries will not work with

WebAssembly. Neither will using C libraries, file system or spawning threads. (Rust and WebAssembly, n.d.)

Rust has a library called wasm-bindgen that provides code for high-level interaction between WebAssembly and JavaScript. It allows JavaScript and WebAssembly to communicate with strings and JavaScript objects, importing JavaScript functionality such as DOM manipulation to Rust, exporting Rust functionality to JavaScript and automatically generating TypeScript bindings for the Rust code used by JavaScript. (The `wasm-bindgen` Guide, n.d.)

According to The Embedded Rust Book (n.d.-b) it is possible to use Rust inside a C or C++ project making the integration to the edge device's codebase feasible. The book also details that C is used for any interoperability between different languages. Java Native Interface should allow using Rust from Java code for the integration with the server codebase. Because the C ABI is used for interoperability, there is an extra step of creating a C-friendly API when compared to just using C/C++ and Emscripten. According to the book the process of building a C API is as follows. First set the cargo build system to output a systems library. Then on any Rust function that is exported outside Rust use the *#[no_mangle]* attributr and mark the functions as *extern "C"*. Also, any data used should conform to C's types. The *#[repr(C)]* on Rust structs guarantees that the Rust compiler uses the same rules as C for organizing data (to The Embedded Rust Book (n.d.-a). The resulting library can then be used with JNI from Java code. Figure 5 shows how Rust would fit into the larger system.

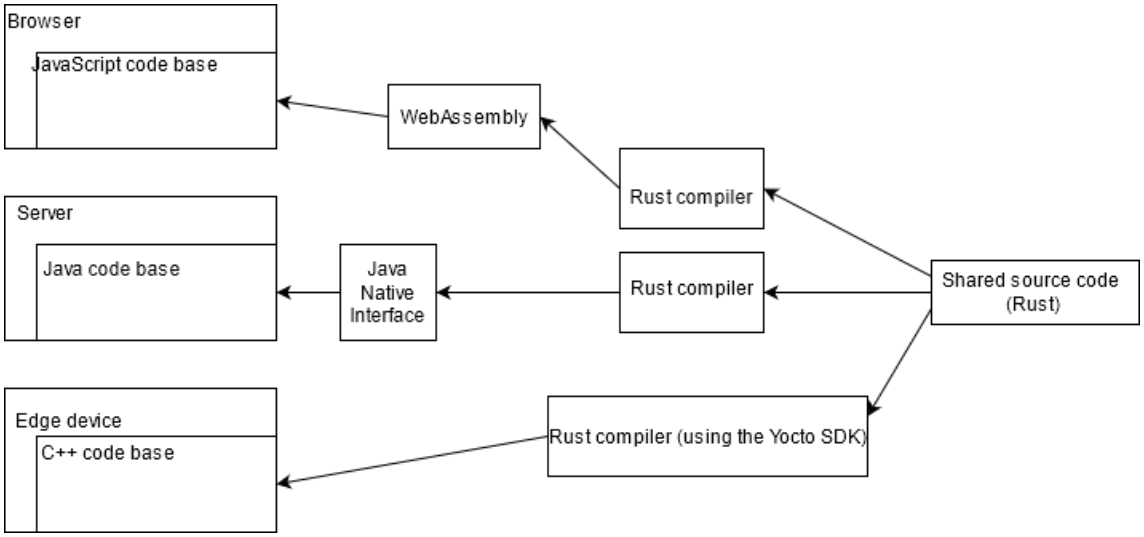

Figure 5 Usage of Rust in the system

Rust's performance on the edge device should be enough as Rust is compiled to the native architecture and does not run on a virtual machine nor use a garbage collector. Rust project does advertise that the language fits well into embedded use cases and is high-performance (Rust Programming Language, n.d.).

On the server and edge device, accessing system resources with Rust would be fine. As noted earlier, on the browser side Rust does not support any system libraries. This in contrast to Emscripten that does emulate the PC environment. This means that any functionality requiring system resources cannot be implemented with the same code on all three platforms. On the other hand, this makes the resulting WebAssembly code much smaller in Rust's case, which is good for the webpage's load times.

Writing generic code for all three platforms with Rust should be possible if certain features such as system libraries are avoided. Like with C/C++ and Emscripten, some platform-specific code is likely needed to integrate to the existing codebases, and there is a small chance of it being larger in the case of Rust.

To summarize: Rust seriously targets WebAssembly support and provides the glue-code for working with the browser environment. It does not emulate system libraries on the

browser but produces small WebAssembly binaries making it a good fit for computational libraries in the web. Rust has also seen production use with WebAssembly and the language's popularity is high currently. Integration to the existing Java and C++ codebases might end up being tricky and require a lot of work.

5.4 AssemblyScript and WebAssembly Micro Runtime

AssemblyScript is a language that targets WebAssembly, while using a TypeScript-like syntax. It integrates to the existing web ecosystem (npm). It is free and open-source software developed by volunteers. AssemblyScript can be described as a TypeScript syntax on top of WebAssembly instructions, statically compiled to produce WebAssembly binaries. AssemblyScript comes with its own JavaScript-like standard library, and memory management and garbage collection runtime. (AssemblyScript, n.d., The AssemblyScript Book, n.d.-b)

WebAssembly Micro Runtime is a standalone WebAssembly runtime. It is compliant to the W3C WebAssembly MVP, has a small runtime size of 85K for the interpreter, and low memory usage. It provides libc support with a built-in libc subset or with WASI (WebAssembly System Interface). It is embeddable with C APIs. It supports multiple architectures including ARMv7 and X86-64 and operating systems like Linux and Windows. (Bytecodealliance/wasm-micro-runtime, n.d.)

AssemblyScript is not as mature as Emscripten and Rust are. In contrast to Emscripten and Rust, AssemblyScript is a young project, has more limited resources and tries to create an alternative from another perspective. AssemblyScript puts anything Web related first and then glues everything together versus Emscripten and Rust trying to lift an existing ecosystem to the web. Binaryen, the compiler infrastructure and toolchain library created by the main Emscipten author, is used by AssemblyScript but it is not as well optimized for AssemblyScript's generated code as it is for LLVM's generated code. (The AssemblyScript Book, n.d.-b)

Because AssemblyScript has its own compiler and different features when compared to JavaScript or TypeScript, it cannot use existing JavaScript or TypeScript libraries. The AssemblyScript documentation lists very few libraries written in AssemblyScript. No real-world commercial projects using AssemblyScript are listed either.

Integrating AssemblyScript to the existing codebases on the server and edge device would be tricky. Figure 6 shows how AssemblyScript and WebAssembly Micro Runtime would fit into the larger system. Bytecodealliance/wasm-micro-runtime (n.d.) does mention that it can be embedded using C API, but it makes no mention of Java leaving it unclear whether it can be integrated into a Java codebase. Likely some method of making it work with the codebases can be found but there is no documentation from the WebAssembly Micro Runtime's side.

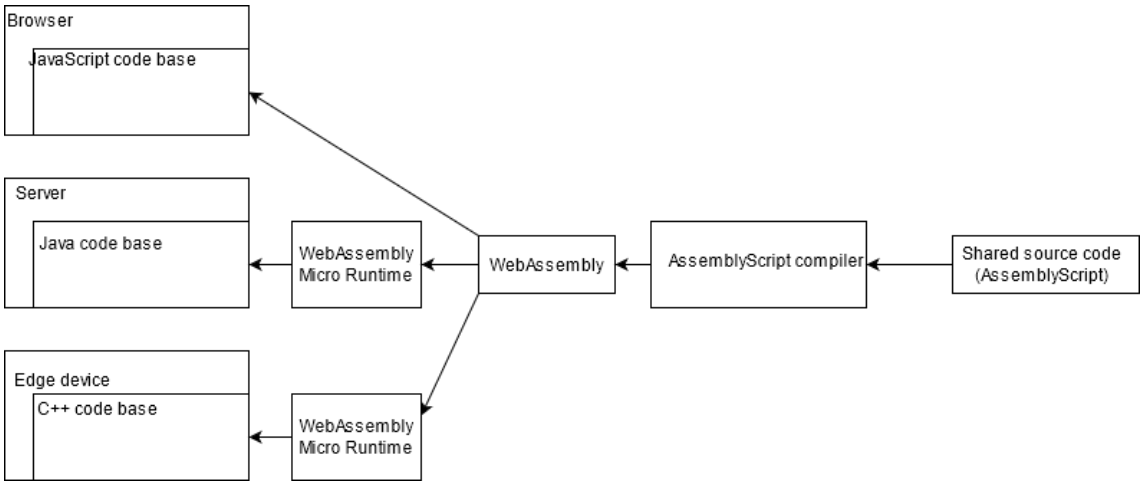

Figure 6 Usage of AssemblyScript and WebAssembly Micro Runtime in the system

The WebAssembly Micro Runtime promises (see Bytecodealliance/wasm-micro-runtime, n.d.) enough performance for the edge device and it even has relatively low memory requirements. It would likely be enough for the case company's needs on the edge device, but proper benchmark would be required before any major decision to use this runtime.

On the browser AssemblyScript does not emulate system resources like Emscripten does. On the server and edge device when running using the WebAssembly Micro Runtime, a library called as-wasi can be used to access some system calls. WASI is an API that provides WebAssembly access to the world outside WebAssembly, AssemblyScript has a low-level WASI set of system calls and as-wasi provides a higher-level API on top of what AssemblyScript provides (As-wasi, n.d.). The documentation of as-wasi is quite barebones and does not give a good idea of what it supports, but for example networking support seems to be missing. Furthermore, WASI appears to be very new and still under development.

In this approach of using AssemblyScript and WebAssembly Micro Runtime, all three platforms would be using the same code. All platforms would be using WebAssembly running on a runtime, instead of server and edge device using code compiled to native architecture and the browser running WebAssembly like the Emscripten and Rust approaches would do. Of course, the integration to the existing codebases would once again require platform-specific code and it is difficult to estimate its extent due to poor documentation of the technologies.

To summarize the findings: AssemblyScript's own documentation admits it is not mature technology. There does not appear to be any large projects using it. The documentation of both AssemblyScript and WebAssembly Micro Runtime is lacking, and these technologies would require extensive testing and benchmarking. This approach of running WebAssembly outside the browser environment is likely too young and bleeding edge still.

5.5 Decision

There are multiple possible approaches based on the requirements defined earlier and the findings from the technologies' documentation.

The main recommended approach is to use JavaScript and the V8 engine. V8 was found to be the most mature JavaScript engine, and it can be embedded into C++ applications or run standalone. Using this approach would allow using a single programming language on all three platforms without compiling to different targets. Platform-specific code would be minimal and only concern the integration to the existing codebases. The performance would be less than a C/C++ implementation would have, but it would not be on a different magnitude making it acceptable for the case company's needs. This approach will be tried out in the next section and the performance will be benchmarked.

The second recommended approach would be to use C/C++ on the server and edge device and to use Emscripten to compile the C/C++ code to WebAssembly for the browser environment. Emscripten was noted to be very mature technology and used in larger projects. C/C++ can be integrated to the existing codebases relatively easy. AssemblyScript and WebAssembly Micro Runtime were simply not mature enough to be used yet. Rust would also be a good choice, but it does not aim to support porting over larger applications to WebAssembly but instead is better fit for small libraries. In this case Rust would very likely still work, but Rust is as a language quite a bit newer than C/C++. In addition, using Rust would introduce a fourth language to the system that already uses JavaScript, Java, and C++.

Third possible approach would be to continue using JavaScript on the browser side and use C/C++ on the server and edge. This would still give the benefit of reducing the languages used for the data processing engine from three to two and give the performance of C/C++ on the edge device. JavaScript is proven to work in the browser environment whereas WebAssembly is new and in development and not that mature of a technology. WebAssembly has been around for a few years but has not had a breakthrough in popularity. This could be because the use case is not useful, or because the technology is not mature or just due to it being new. Using JavaScript would be in a sense a safe choice. The current functionality on the browser side is intertwined with the UI and JavaScript is perfect for that. On the other hand, if V8 engine has acceptable performance on the

edge device, then this approach is more work and not that beneficial, which is why it is not recommended. Finally, this approach would not answer the requirement of using a single implementation.

6 Proof-of-concept

In this section the implementation of a proof-of-concept using JavaScript with V8 is presented. The goal is to find out if it is feasible to use JavaScript with V8 on the case company's edge device. If the performance is significantly (i.e., on a different magnitude) worse than C++'s, then alternative approaches such as C++ with Emscripten are recommended instead of JavaScript and V8. Also, the memory usage and startup time of the V8 engine are considered.

In this phase the performance of the V8 engine is benchmarked when running JavaScript code implementing a few blocks of the data processing engine. The implementation of the blocks will be simplified; only the core functionality will be tested, and the interface will be simple. Any data input and output through the network or combining logic blocks will not be considered. The JavaScript's performance will be compared to the implementation of the same blocks in C++. The device used is the case company's edge device.

First the V8 JavaScript engine must be compiled into a static library so that it can be embedded in a C++ application. Cross-compilation using the Yocto SDK is required, as the edge device is ARM-based, and its Linux is developed using the Yocto toolchain. Then a test application is written in C++. The test application will measure the performance of JavaScript when running the code in the V8 engine embedded in the application and measure the performance of C++ implementations of the same logic blocks.

6.1 Implementation

In this chapter the implementation of the proof-of-concept is detailed. This includes the compilation of the V8 into a static library, the coding of a few basic logic blocks in JavaScript and C++, and the setup for embedding V8 into the code. In the case that V8 is taken into use, this document may serve as a guide for the real implementation of the system.

6.1.1 Compiling V8

For this phase, a VirtualBox virtual machine using the Linux distribution Ubuntu 20.04 was created. All the following steps were done in this environment.

According to V8.dev (n.d.-a), V8 is built using a tool called GN which is a meta build system that generates build files for other build systems. V8.dev (n.d.-a) and Depot_tools_tutorial(7) (n.d.) document how to get the required files downloaded and installed, and how to build the V8 engine. Additional details for the configuration were found from Stackoverflow.com, various blog posts and by trying out different flags. The whole process is depicted in Figure 7.

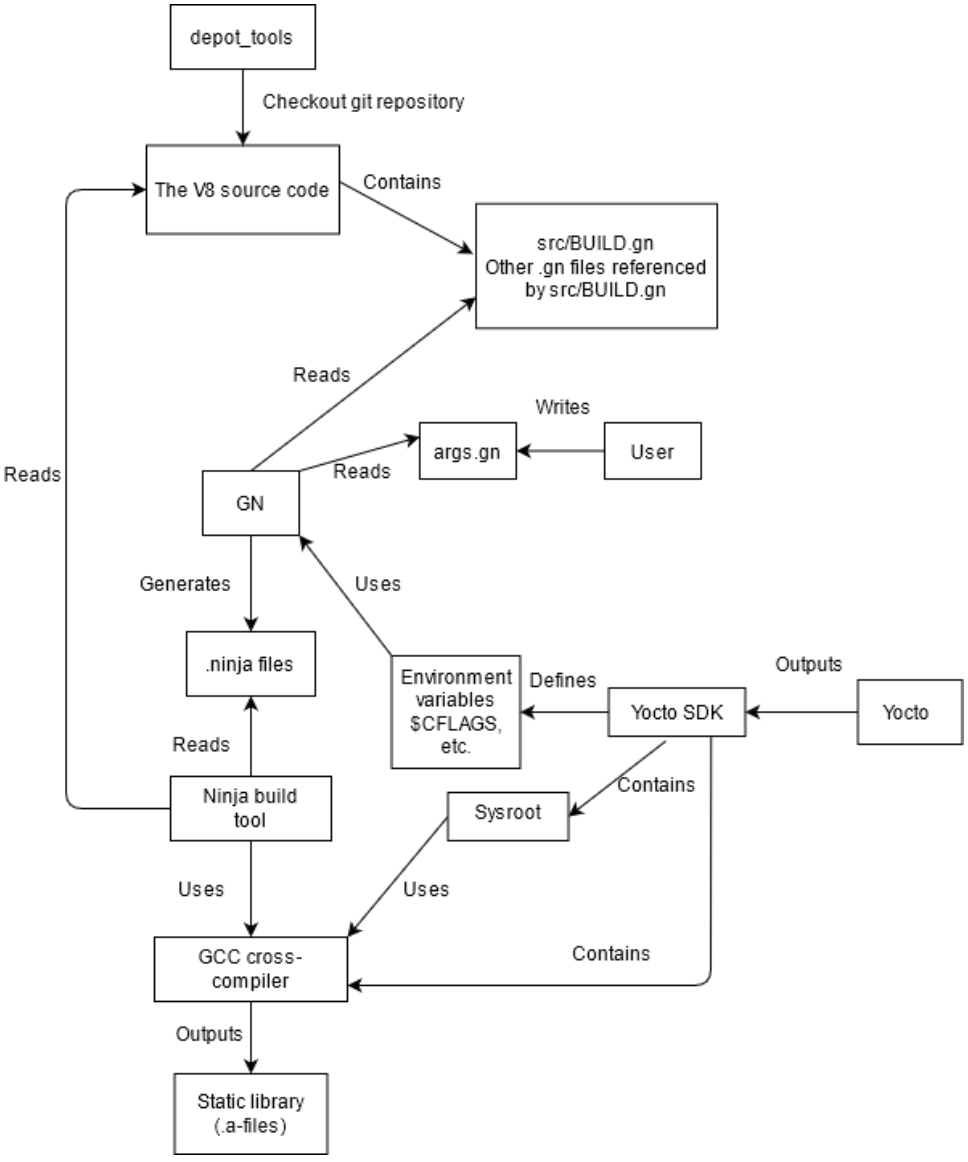

Figure 7 The process of cross-compiling the V8 engine and relationships between various tools and files.

First the following packages were installed using *apt:* python 2.7, Git and gcc-multilib. A git extension called depot_tools (https://chromium.googlesource.com/chromium/tools/depot_tools.git) was required for checking out the V8 git repository. After adding depot_tools' directory to the path environment variable, the V8 source code could be checked out with the *fetch v8* command from depot_tools.

The latest stable release of V8 at the time, version 8.8, was used by checking out the corresponding Git branch (branch-heads/8.8) in the source directory. The build

dependencies were installed by running a script with flags for opting to install 32-bit and Arm specific dependencies.

```
./build/install-build-deps.h –lib32 –arm
gclient sync
```

To cross-compile for the edge device, the Yocto SDK had to be used. The SDK was obtained as an output from the Yocto setup that had been used to build the edge device's Linux OS. This step had been done previously and will not be detailed here. The SDK was distributed as an auto-extracting script file and executing that script file resulted in the SDK being installed. The output contains a script, and it defines environment variables such as *$CXX*, which can be used to use the SDK's g++ cross-compiler, and *$CFLAGS*, which sets the correct flags for the compiler. The environment script file was used with the *source* command.

Then the GN build tool had to be configured to use this SDK. In the V8 source directory, the file *./tools/toolchain/BUILD.gn* was modified to contain a toolchain definition for the SDK

```
gcc_toolchain("yoctosdk") {

  cc = getenv("CC")
  cxx = getenv("CXX")

  readelf = getenv("READELF")
  nm = getenv("NM")
  ar = getenv("AR")
  ld = cxx
  toolchain_args = {
    current_cpu = "x64"
    current_os = "linux"
    is_clang = false
  }
}
```

The GN tool generates the build files with the command

```
gn gen out/arm
```

and the arguments can be set with

```
gn args out/arm
```

The following arguments were given:

```
custom_toolchain = "//tools/toolchain:yoctosdk"
target_cpu = "arm"
target_os = "linux"
target_sysroot = "/opt/poky/3.1.1/sysroots/cortexa5t2hf-vfp-
poky-linux-gnueabi"
is_clang = false
is_component_build = false
use_gold = false
v8_monolithic = true
v8_use_external_startup_data = false
use_custom_libcxx = false
use_goma = false
goma_dir = "None"
v8_static_library = true
is_debug = false
arm_use_neon = false
```

Now the build files for the Ninja build tool were generated and the V8 could be compiled with the *autoninja* command

```
autoninja -C out/arm v8_monolith
```

resulting in the static V8 library as an output. To compile the sample code for embedding the V8 in a C++ program, the following command was used

```
$CXX -I. -Iinclude samples/hello-world.cc -o
hello_world_arm -lv8_monolith -Lout/arm/obj -pthread
-std=c++14 $CXXFLAGS
```

The sample program was run on the case company's edge device to verify the cross-compilation was successful.

6.1.2 Block selection

The average, sin and round blocks were selected to be implemented in this proof-of-concept. The average block simply counts the average of the input array, the sin block performs the sine function on all elements of the input array and the round block rounds all elements of the input array. These were selected because they are simple blocks that perform numerical operations making them a good fit for simple benchmarking. Also, a block that returns the first item of the input array, and a block that performs an empty function call were implemented to compare the overhead in function calls and accessing arrays in JavaScript versus C++.

The input data for all blocks was the same – 100 000 floating point values were generated with the Python script detailed below and written to a file.

```
import random

with open("data.txt", "w") as f:
    for i in range(100000):
        f.write(str(random.random() * 1000))
        f.write(" ")
```

6.1.3 JavaScript implementation

In the JavaScript implementation, the blocks were implemented as simple JavaScript functions as seen below in a file called *blocks.js*. The functions take a JavaScript array containing the previously generated test data as an input.

```
// return the average value of array input
function average(input) {
    var count = input.length;
    var sum = 0;
    for (var i = 0; i < count; i++) {
        sum += input[i];
    }
    return sum / count;
}

//return an array of the sines of the array input
function sin(input) {
    var output = [];
```

```
    var count = input.length;
    for (var i = 0; i < count; i++) {
        output[i] = Math.sin(input[i]);
    }
    return output;
}

//return an array of input array's values rounded to preci-
sion
function round(input, precision) {
    var output = [];
    var count = input.length;
    var n = Math.pow(10, precision);
    for (var i = 0; i < count; i++) {
        output[i] = Math.round(input[i] * n) / n;
    }
    return output;
}

function emptyFunctionCall() {
    return;
}

function firstItem(input) {
    return input[0];
}
```

6.1.4 C++ implementation

The C++ implementation likewise implemented the blocks as simple C++ functions. As usual for C++, a header file *blocks.h* with the function declarations was created.

```
#include <vector>
#include <math.h>

double average(std::vector<double>&);
std::vector<double> calcSin(std::vector<double>&);
std::vector<double> calcRound(std::vector<double>&, int );
void emptyFunctionCall();
double firstItem(std::vector<double>&);
```

The functions, implemented in *blocks.cpp*, take the previously generated test data as an input. The input is given as a *vector* from the standard library. Vectors were chosen because they are variable-length just as JavaScript arrays are. The functions were implemented to be as similar to the JavaScript implementations as possible. The function

emptyFunctionCall included the assembly directive *"nop"* to avoid the compiler optimizing the empty function call away.

```
#include "blocks.h"

double average(std::vector<double> &input) {
    int count = input.size();
    double sum = 0;
    for (int i = 0; i < count; i++) {
        sum += input[i];
    }
    return sum / count;
}

std::vector<double> calcSin(std::vector<double> &input) {
    std::vector<double> output = std::vector<double>();
    int count = input.size();
    for (int i = 0; i < count; i++) {
        output.push_back(sin(input[i]));
    }
    return output;
}

std::vector<double> calcRound(std::vector<double> &in-
put, int precision) {
    std::vector<double> output = std::vector<double>();
    int count = input.size();
    int n = pow(10, precision);
    for (int i = 0; i < count; i++) {
        output.push_back(round(input[i] * n) / n);
    }
    return output;
}

void emptyFunctionCall() {
    asm("nop");
}

double firstItem(std::vector<double> &input) {
    return input[0];
}
```

6.2 Performance testing

A test application was written in C++. The goal was to measure the execution times of the previously implemented logic blocks, and the startup time of the V8 engine.

6.2.1 The test application

The test application was written in C++. It initializes the V8 JavaScript engine and reads the input data from a text file constructing the input *vector<double>* for the C++ implementation and the JavaScript array *v8::Local<v8::Array>* for the JavaScript implementations. The time spent reading the input data and constructing the input vector and array were not included in the benchmarks, but the time spent initializing the V8 engine was measured. After the initializations and data preparation, the test application ran each of the JavaScript and C++ functions and measured the execution time of each function. Time was measure with the *std::chrono::steady_clock* clock. The code of the test application is detailed in Appendix 1.

In order to test the application with both *-O0* and *-O3* optimization levels, it was cross-compiled with

```
$CXX -I. -Iinclude v8poc.cpp blocks.cpp -o v8poc -lv8_mono-
lith -Lout/arm/obj -pthread -std=c++14 $CXXFLAGS -Wno-psabi
-O0
```

and

```
$CXX -I. -Iinclude v8poc.cpp blocks.cpp -o v8poc -lv8_mono-
lith -Lout/arm/obj -pthread -std=c++14 $CXXFLAGS -Wno-psabi
-O3
```

Optimization levels *-O0* and *-O3* were both tested. The cross-compilation was done using the Yocto SDK. The cross-compiled executables were then run on the case company's edge device.

The memory usage of the executable was measured with the external commands *pmap -x <pid>* and *cat /proc/<pid>/status* while running the executable.

6.2.2 Results

The test application was run three times with both optimization levels *-O0* and *-O3*. The results were gathered into Table 3.

Table 3 The results of the benchmark

	O0			O3		
Time in milliseconds	1	2	3	1	2	3
V8 startup	64.4497	61.6588	60.6782	62.5851	60.7518	59.4027
JavaScript average	172.328	166.562	166.797	167.371	165.738	166.152
JavaScript sin	236.048	232.766	289.57	232.599	234.466	237.075
JavaScript round	210.268	274.42	220.971	213.383	208.659	208.649
JavaScript empty function call	0.932606	0.673819	0.500848	0.647697	0.499333	0.50303
JavaScript access first item	0.692546	0.502546	0.489031	0.492666	0.489939	0.487394
C++ average	10.2786	10.0273	10.3802	2.23782	2.36564	2.23976
C++ sin	121.602	120.579	121.833	73.9808	71.0308	70.5568
C++ round	98.71	98.5861	98.1158	44.5305	44.7596	47.242
C++ empty function call	0.003091	0.003273	0.002	0.002727	0.003576	0.003515
C++ access first item	0.002545	0.003334	0.002606	0.002425	0.002909	0.002848

The output of the *pmap -x <pid>* command was

```
~# pmap -x 13466
13466:   ./v8poc
Address   Kbytes     RSS   Dirty Mode  Mapping
00446000   11112    5928    5928 r-x-- v8poc
00f30000     180     180     180 r---- v8poc
00f5d000      60      60      60 rw--- v8poc
00f6c000     708     576     576 rw---   [ anon ]
21d40000      48      48      48 rw---   [ anon ]
24cc0000      12      12      12 rw---   [ anon ]
25700000     256     256     256 rw---   [ anon ]
39980000     256      68      68 rw---   [ anon ]
40480000     256     168     168 rw---   [ anon ]
408c0000     256     256     256 rw---   [ anon ]
4d380000      12      12      12 rw---   [ anon ]
4d383000       4       0       0 -----   [ anon ]
4d384000      64      64      64 r-x--   [ anon ]
4d394000       4       4       0 -----   [ anon ]
529c0000     256      24      24 rw---   [ anon ]
55c00000     256     256     256 rw---   [ anon ]
5ce40000     124     124     124 r----   [ anon ]
5d5c0000     400     400     400 rw---   [ anon ]
b6270000    1028     784     784 rw---   [ anon ]
b6454000       4       0       0 -----   [ anon ]
b6455000    8192       8       8 rw---   [ anon ]
b6c55000     884     532       0 r-x-- libc-2.31.so
b6d32000      60       0       0 ----- libc-2.31.so
b6d41000       8       8       8 r---- libc-2.31.so
```

```
b6d43000       8       8       8 rw--- libc-2.31.so
b6d45000       8       8       8 rw---   [ anon ]
b6d47000      72      72       0 r-x-- libpthread-2.31.so
b6d59000      60       0       0 ----- libpthread-2.31.so
b6d68000       4       4       4 r---- libpthread-2.31.so
b6d69000       4       4       4 rw--- libpthread-2.31.so
b6d6a000       8       4       4 rw---   [ anon ]
b6d6c000      96      56       0 r-x-- libgcc_s.so.1
b6d84000      64       0       0 ----- libgcc_s.so.1
b6d94000       4       4       4 r---- libgcc_s.so.1
b6d95000       4       4       4 rw--- libgcc_s.so.1
b6d96000     348      52       0 r-x-- libm-2.31.so
b6ded000      60       0       0 ----- libm-2.31.so
b6dfc000       4       4       4 r---- libm-2.31.so
b6dfd000       4       4       4 rw--- libm-2.31.so
b6dfe000    1180     732       0 r-x-- libstdc++.so.6.0.28
b6f25000      60       0       0 ----- libstdc++.so.6.0.28
b6f34000      28      28      28 r---- libstdc++.so.6.0.28
b6f3b000       4       4       4 rw--- libstdc++.so.6.0.28
b6f3c000       4       4       4 rw---   [ anon ]
b6f3d000     100     100       0 r-x-- ld-2.31.so
b6f62000      16      16      16 rw---   [ anon ]
b6f66000       4       4       4 r---- ld-2.31.so
b6f67000       4       4       4 rw--- ld-2.31.so
bef71000     132      12      12 rw---   [ stack ]
befb0000       4       0       0 r-x--   [ anon ]
ffff0000       4       0       0 r-x--   [ anon ]
-------- ------- ------- -------
total kB   26728   10896    9348
```

And the output of the *cat /proc/<pid>/status* command was

```
~# cat /proc/13466/status
Name:   v8poc
Umask:  0022
State:  S (sleeping)
Tgid:   13466
Ngid:   0
Pid:    13466
PPid:   2029
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
NStgid: 13466
NSpid:  13466
NSpgid: 13466
NSsid:  2029
VmPeak:    25660 kB
VmSize:    25296 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      8724 kB
```

```
VmRSS:      8724 kB
RssAnon:             1336 kB
RssFile:             7388 kB
RssShmem:               0 kB
VmData:    10632 kB
VmStk:       132 kB
VmExe:     11112 kB
VmLib:      2748 kB
VmPTE:        40 kB
VmSwap:        0 kB
CoreDumping:    0
THP_enabled:    0
Threads:        2
SigQ:   0/1430
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000004
SigCgt: 0000000180000000
CapInh: 0000000000000000
CapPrm: 0000003fffffffff
CapEff: 0000003fffffffff
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000
NoNewPrivs:     0
Speculation_Store_Bypass:       unknown
Cpus_allowed:   1
Cpus_allowed_list:      0
voluntary_ctxt_switches:        1
nonvoluntary_ctxt_switches:     31
```

6.3 Discussion

As was expected, the benchmarking showed that the C++ implementations performed better than JavaScript and V8. The uncertainty was whether the JavaScript performance was on an unacceptable level.

The V8 startup time was around 60 milliseconds, which is not too much. If the V8 engine is kept running continuously, the startup delay is only suffered when rebooting the device.

If looking at the *-O3* performance in Table 3, the JavaScript implementations were from 5 to almost 200 times slower than the C++ implementations. When taking into account that the input data was 100 000 floating point values, another way of interpreting the

results is that the V8 engine can round 500 000 values each second if the device does nothing else or round 50 000 values each 100 millisecond interval. 100 milliseconds could be the lowest typical control loop length on the edge device. The overhead of calling JavaScript functions was significant (200x) as seen from the execution times of calling empty functions. This overhead could be reduced by designing the architecture so that calls between C++ and JavaScript were minimized. For example the combination of the blocks could be done in JavaScript so that only a single JavaScript call from C++ is required to process a single logic flow.

The memory consumption of the test application was around 25 000 to 27 000 kB. Of this amount most of it can be assumed to be caused by the V8 engine. This amount still fits into the edge device's memory, although the static consumption by the V8 engine is quite a large portion of the edge device's free memory (256 MB in total).

This test and benchmarking shows that using V8 and JavaScript on the case company's edge device is indeed feasible. The performance proved to be significantly worse than C++'s as was expected. On the other hand, the performance was not bad enough to completely rule out the possibility of using JavaScript and V8. If the amount of data and calculations being processed is not too large, V8 and JavaScript will be fine for the use case.

7 Conclusions

Multi-platform software simplifies the process of developing software for multiple targets. Developing only a single implementation instead of multiple implementations, saves time and resources both in development and maintenance of the software. Guarantees of conforming behavior across platforms can only be achieved with multi-platform code – with multiple implementations there is always some uncertainty whether the behavior is the same. To get all these benefits, a flexible and high-performance technology is required.

The literature review showed that previous research has found the same problems as the case company has faced. The problem of multi-platform software has been solved in various ways in many contexts, using for example compilers, interpreters, web applications and cross-platform mobile GUI SDKs. A lot of the existing research focuses on multi-platform web and mobile software as they are widespread currently, but the use case is like the one in this thesis. The study showed that multi-platform software is required in complex software applications targeting different kinds of hardware and software platforms.

This thesis gave a recommendation to the case company. The recommendation consists of using JavaScript for the implementation of the data processing engine and using the V8 JavaScript engine on the server and edge devices to execute JavaScript. A set of technologies was reviewed, and this approach was chosen because of the maturity of the technology, its performance, its easy integration to the existing codebases and likelihood that it will receive support in the future. The requirement of sharing code base with the edge device is driven by the need for flexibility and independence from internet connection. The purpose is not to do real-time computation. Thus, JavaScript and V8 solve the issue.

This approach also allows using an existing implementation of the software, or at the very least large parts of it. The second alternative of using C/C++ and Emscripten is

recommended in case the first approach proves to be problematic in the real implementation. The recommendation is specific to the case company and the particular software system but can be used to guide decisions in other cases too.

The limited choice in technologies that can be used in the web browser was found to be the most limiting factor for the technology choice. Also, the performance requirements and limited hardware of the edge devices used by the case company further constrained the choice. Surprisingly, the ARM architecture of the edge device also ruled out certain technologies. Finally, due to the other requirements the choice was between only a few technologies. The others were ruled out due to the lacking maturity of the technology or being niche or hobby projects. As the review was based on the technologies' documentation, it is plausible that the projects were more mature than they seemed and only their documentation was lacking leading into a false conclusion. However, lacking documentation is a major negative point.

The study also presented some of the limitations and constraints of the chosen technology in the review process. A proof-of-concept was developed and compared with a competing approach. The performance of this proof-of-concept was tested and documented. The technology was found to have adequate performance on all target platforms and its limitations and constraints did not make it impossible to use on any of the platforms.

In the research process some general observations were made. When designing multi-platform software, a key issue is that a constraint on one of the platforms will apply to all the platforms. For example, in this thesis the constraint imposed by the web browser environment narrowed down the choice of technology to those that support JavaScript or WebAssembly. It was discovered that even with a technology that allows cross-platform code, there might still be a need for some platform-specific code. Also, it was noted that an important factor to consider when choosing a cross-platform technology is the possibility of integrating it to existing code bases.

These general observations should be considered when planning, designing, and developing multi-platform software. It is important to consider these issues early in the software development process to avoid large refactoring and integration processes or writing large amounts of new code. Depending on the platforms and technologies, it might not be realistic to expect to run the same code on all platforms without any platform-specific code.

During the technology review and proof-of-concept implementation, multiple new research questions and ideas for future research were found. As this thesis only presented a proof-of-concept, many facets of the technology choice were left unexplored. Interfacing with other programs and software was not extensively evaluated. Using the V8 engine with Java code was not explored. If the V8 engine is taken into use, the process of using it in the existing Java code base will need to be researched.

Because WebAssembly is relatively new technology, it has not been researched extensively and more testing is required in case it is chosen to be taken into use. Since WebAssembly was not tested, it is also impossible to say if it would have had better performance on the edge devices or the browser than JavaScript.

JavaScript engines other than V8 were not evaluated. Although prior research suggested their performance is inferior to V8's, it is not certain that is the case on the case company's device. These alternative JavaScript engines would have smaller memory footprint, which could make them interesting alternatives to explore in the future.

Prior research with regards to targeting multiple different platforms, such was the case in this thesis, was found lacking. No research or discussions detailing cross-platform software across the browser, server and edge device environments were found. This study produced clear answers for the case company's problem area, and it compiled information in a new way. The results can be used for solving similar problems in general.

References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, techniques, & tools (2nd ed). Pearson/Addison Wesley.

Andreessen, M. (1998, June 24). TechVision: Innovators of the Net: Brendan Eich and JavaScript. https://web.archive.org/web/20080208124612/http://wp.netscape.com/comprod/columns/techvision/innovators_be.html

AssemblyScript. (n.d.). Retrieved February 7, 2021, from https://www.assemblyscript.org/

As-wasi. (n.d.). Retrieved February 7, 2021, from https://github.com/jedisct1/as-wasi (Original work published 2019)

Bellard, F. (n.d.). QuickJS Benchmark. Retrieved February 8, 2021, from https://bellard.org/quickjs/bench.html

Bishop, J., & Horspool, N. (2006). Cross-Platform Development: Software that Lasts. Computer, 39(10), 26–35. https://doi.org/10.1109/MC.2006.337

Bytecode Alliance. (n.d.). Bytecode Alliance. Retrieved February 3, 2021, from https://bytecodealliance.org/

Bytecodealliance/wasm-micro-runtime. (n.d.). Retrieved January 29, 2021, from https://github.com/bytecodealliance/wasm-micro-runtime

Bytecodealliance/wasmtime. (n.d.). GitHub. Retrieved February 21, 2021, from https://github.com/bytecodealliance/wasmtime

Cesanta/mjs. (n.d.). Retrieved February 14, 2021, from https://github.com/cesanta/mjs

Chromium Blog. (2019, June 20). WebAssembly brings Google Earth to more browsers. Chromium Blog. https://blog.chromium.org/2019/06/webassembly-brings-google-earth-to-more.html

Cusumano, M. A., & Yoffie, D. B. (1999). What Netscape learned from cross-platform software development. Communications of the ACM, 42(10), 72–78. https://doi.org/10.1145/317665.317678

Depot_tools_tutorial(7). (n.d.). Retrieved February 28, 2021, from https://commondatastorage.googleapis.com/chrome-infra-docs/flat/depot_tools/docs/html/depot_tools_tutorial.html#_setting_up

Duktape. (n.d.). Retrieved February 14, 2021, from https://duktape.org/

Electronjs.org. (n.d.). Electron | Build Cross-Platform Desktop Apps with JavaScript, HTML, and CSS. Retrieved March 7, 2021, from https://www.electronjs.org/

Emscripten 2.0.12 documentation. (n.d.-a). About Emscripten. https://emscripten.org/docs/introducing_emscripten/about_emscripten.html

Emscripten 2.0.12 documentation. (n.d.-b). API Limitations. https://emscripten.org/docs/porting/guidelines/api_limitations.html#

Emscripten 2.0.12 documentation. (n.d.-c). API Reference. https://emscripten.org/docs/api_reference/index.html

Emscripten 2.0.12 documentation. (n.d.-d). Building Projects. https://emscripten.org/docs/compiling/Building-Projects.html

Fitzgerald, N. (2018, January 18). Oxidizing Source Maps with Rust and WebAssembly – Mozilla Hacks—The Web developer blog. Mozilla Hacks – the Web Developer Blog. https://hacks.mozilla.org/2018/01/oxidizing-source-maps-with-rust-and-webassembly

Free Software Foundation. (n.d.). Using the GNU Compiler Collection (GCC). Retrieved January 30, 2021, from https://gcc.gnu.org/onlinedocs/gcc-10.2.0/gcc/

Frequently Asked Questions · The Rust Programming Language. (2016, June 9). https://web.archive.org/web/20160609195720/https://www.rust-lang.org/faq.html#project

Gartner. (2017, May 23). Gartner Says Worldwide Sales of Smartphones Grew 9 Percent in First Quarter of 2017. https://www.gartner.com/en/newsroom/press-releases/2017-05-23-gartner-says-worldwide-sales-of-smartphones-grew-9-percent-in-first-quarter-of-2017

Gosling, J., & McGilton, H. (1996, May). The Java Language Environment. https://www.oracle.com/java/technologies/introduction-to-java.html

Hogg, S. (2014, May 26). Software Containers: Used More Frequently than Most Realize. Network World. https://www.networkworld.com/article/2226996/software-containers--used-more-frequently-than-most-realize.html

Horn, D. R. (n.d.). Building better compression together with DivANS. Retrieved February 7, 2021, from https://dropbox.tech/infrastructure/building-better-compression-together-with-divans

Ierusalimschy, R., Figueiredo, L. H. de, & Filho, W. C. (1996). Lua—An Extensible Extension Language. Software: Practice and Experience, 26(6), 635–652. https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P Retrieved February 21, 2021 from https://www.lua.org/spe.html

Java Native Interface Specification. (n.d.). Chapter 1: Introduction. Retrieved February 21, 2021, from https://docs.oracle.com/en/java/javase/15/docs/specs/jni/intro.html

Java Scripting Programmer's Guide. (n.d.). Retrieved February 21, 2021, from https://docs.oracle.com/javase/7/docs/technotes/guides/scripting/programmer_guide/

JEP 372: Remove the Nashorn JavaScript Engine. (n.d.). Retrieved February 21, 2021, from https://openjdk.java.net/jeps/372

Jerryscript.net. (n.d.). Retrieved February 14, 2021, from https://jerryscript.net/

Kernighan, B., & Ritchie, D. (1988). The C Programming Language (2nd ed.). Prentice Hall PTR.

MDN Web Docs. (2021a). About JavaScript—JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript

MDN Web Docs. (2021b). WebAssembly Concepts—WebAssembly | MDN. WebAssembly Concepts. https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts

Migration Guide from Nashorn to GraalVM JavaScript. (n.d.). Retrieved February 21, 2021, from https://www.graalvm.org/reference-manual/js/NashornMigrationGuide/

Mozilla. (n.d.). What Is a Web Browser? Retrieved February 21, 2021, from https://www.mozilla.org/en-US/firefox/browsers/what-is-a-browser/

Oliveira, F., & Mattos, J. (2020). Analysis of WebAssembly as a Strategy to Improve JavaScript Performance on IoT Environments. Anais Estendidos Do Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC), 133–138. https://doi.org/10.5753/sbesc_estendido.2020.13102

Oracle/graaljs. (n.d.). GitHub. Retrieved February 21, 2021, from https://github.com/oracle/graaljs

Pack, S. (2018, October 16). Serverless Rust with Cloudflare Workers. The Cloudflare Blog. https://blog.cloudflare.com/cloudflare-workers-as-a-serverless-rust-platform/

PC Magazine. (2021). Definition of cross platform. PCMAG. https://www.pcmag.com/encyclopedia/term/cross-platform

Rust and WebAssembly. (n.d.). Which Crates Will Work Off-the-Shelf with WebAssembly? - Rust and WebAssembly. Retrieved February 7, 2021, from https://rustwasm.github.io/docs/book/reference/which-crates-work-with-wasm.html

Rust Programming Language. (n.d.). Retrieved February 4, 2021, from https://www.rust-lang.org/

Shi, W., & Dustdar, S. (2016). The Promise of Edge Computing. Computer, 49(5), 78–81. https://doi.org/10.1109/MC.2016.145

Sinha, A. (1992). Client-server computing. Communications of the ACM, 35(7), 77–98. https://doi.org/10.1145/129902.129908

Soquet, P. (2017, May 24). XS7 @ TC-39. https://www.moddable.com/XS7-TC-39.php

Stack Overflow Developer Survey 2019. (2019a). Stack Overflow. https://insights.stackoverflow.com/survey/2019#technology-_-other-frameworks-libraries-and-tools

Stack Overflow Developer Survey 2019. (2019b). Stack Overflow. https://insights.stackoverflow.com/survey/2019/#technology-_-most-loved-dreaded-and-wanted-languages

Stroustrup, B. (2020). Thriving in a crowded and changing world: C++ 2006–2020. Proceedings of the ACM on Programming Languages, 4(HOPL), 1–168. https://doi.org/10.1145/3386320

Taivalsaari, A., Mikkonen, T., Ingalls, D., & Palacz, K. (2008). Web Browser as an Application Platform. 2008 34th Euromicro Conference Software Engineering and Advanced Applications, 293–302. https://doi.org/10.1109/SEAA.2008.17

The `wasm-bindgen` Guide. (n.d.). Introduction. Retrieved February 7, 2021, from https://rustwasm.github.io/docs/wasm-bindgen/

The AssemblyScript Book. (n.d.-a). Frequently Asked Questions. https://www.assemblyscript.org/frequently-asked-questions.html

The AssemblyScript Book. (n.d.-b). Introduction. https://www.assemblyscript.org/introduction.html

The Embedded Rust Book. (n.d.-a). A Little C with Your Rust. https://rust-embedded.github.io/book/interoperability/c-with-rust.html

The Embedded Rust Book. (n.d.-b). A Little Rust with Your C. https://rust-embedded.github.io/book/interoperability/rust-with-c.html

V8.dev. (n.d.-a). Building V8 with GN · V8. https://v8.dev/docs/build-gn

V8.dev. (n.d.-b). V8 JavaScript Engine. https://v8.dev/

Wagner, L. (2017, November 21). WebAssembly Will Finally Let You Run High-Performance Applications in Your Browser—IEEE Spectrum. IEEE Spectrum: Technology, Engineering, and Science News. https://spectrum.ieee.org/computing/software/webassembly-will-finally-let-you-run-highperformance-applications-in-your-browser

WebAssembly.org. (n.d.-a). https://webassembly.org/

WebAssembly.org. (n.d.-b). FAQ - WebAssembly. https://webassembly.org/docs/faq/

WebAssembly.org. (n.d.-c). I Want To… - WebAssembly. https://webassembly.org/getting-started/developers-guide/

WebAssembly.org. (n.d.-d). Use Cases - WebAssembly. https://webassembly.org/docs/use-cases/

WebAssembly.org. (n.d.-e). WebAssembly High-Level Goals - WebAssembly. https://webassembly.org/docs/high-level-goals/

Appendices

Appendix 1. The test application code

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <chrono>

#include "include/v8.h"
#include "include/libplatform/libplatform.h"
#include "blocks.h"

int main(int argc, char* argv[]) {

    //start measuring V8 startup time
    auto c_start = std::chrono::steady_clock::now();
    v8::V8::InitializeICUDefaultLocation(argv[0]);
    v8::V8::InitializeExternalStartupData(argv[0]);
    std::unique_ptr<v8::Platform> platform = v8::plat-
form::NewDefaultPlatform();
    v8::V8::InitializePlatform(platform.get());
    v8::V8::Initialize();

    v8::Isolate::CreateParams createParams;
    createParams.array_buffer_allocator = v8::Array-
Buffer::Allocator::NewDefaultAllocator();
    v8::Isolate* isolate = v8::Isolate::New(createParams);

    auto c_end = std::chrono::steady_clock::now();
    std::chrono::duration<dou-
ble, std::milli> elapsed = c_end - c_start;
    auto V8StartupTime = elapsed.count();

    {
        v8::Isolate::Scope isolate_scope(isolate);
        v8::HandleScope handle_scope(isolate);

        v8::Local<v8::Context> context = v8::Con-
text::New(isolate);
        v8::Context::Scope context_scope(context);

        //read javascript file and run the script
        //this makes the javascript functions available
        std::ifstream t("./blocks/blocks.js");
        std::stringstream buffer;
        buffer << t.rdbuf();
        t.close();
        v8::Local<v8::String> source = v8::String::New-
FromUtf8(isolate, buffer.str().c_str()).ToLocalChecked();
```

```
        v8::Local<v8::Script> script = v8::Script::Com-
pile(context, source).ToLocalChecked();
        v8::TryCatch tryCatch(isolate);
        v8::MaybeLocal<v8::Value> result = script->Run(con-
text);
        if (result.IsEmpty()) {
            v8::String::Utf8Value e(isolate, tryCatch.Ex-
ception());
            std::cerr << "Exception: " << *e << std::endl;
        }

        //read test data from file
        std::fstream myfile("./data.txt", std::ios_base::in
);
        double val;
        std::vector<double> data = std::vector<double>();
        while (myfile >> val)
        {
            data.push_back(val);
        }
        myfile.close();

        //copy data into a javascript array
        v8::Local<v8::Array> jsdata = v8::Array::New(iso-
late, data.size());
        for (int i = 0; i < data.size(); i++) {
            v8::Local<v8::Value> num = v8::Number::New(iso-
late, data[i]);
            v8::Maybe<bool> res = jsdata->Set(con-
text, i, num);
        }

        v8::Local<v8::Object> global = context->Global();
        //get the js functions
        v8::Local<v8::Function> averageFunc = v8::Lo-
cal<v8::Function>::Cast(
            global->Get(context, v8::String::New-
FromUtf8(isolate, "average").ToLocalChecked()).ToLocal-
Checked()
            );
        v8::Local<v8::Function> sinFunc = v8::Lo-
cal<v8::Function>::Cast(
            global->Get(context, v8::String::New-
FromUtf8(isolate, "sin").ToLocalChecked()).ToLocalChecked()
            );
        v8::Local<v8::Function> roundFunc = v8::Lo-
cal<v8::Function>::Cast(
            global->Get(context, v8::String::New-
FromUtf8(isolate, "round").ToLocalChecked()).ToLocal-
Checked()
            );
        v8::Local<v8::Function> emptyFunc = v8::Lo-
cal<v8::Function>::Cast(
```

```
            global->Get(context, v8::String::New-
FromUtf8(isolate, "emptyFunctionCall").ToLocal-
Checked()).ToLocalChecked()
            );
        v8::Local<v8::Function> firstItemFunc = v8::Lo-
cal<v8::Function>::Cast(
            global->Get(context, v8::String::New-
FromUtf8(isolate, "firstItem").ToLocalChecked()).ToLocal-
Checked()
            );
        //arguments to pass to the javascript functions
        v8::Local<v8::Value> args1[1];
        args1[0] = jsdata;
        v8::Local<v8::Value> args2[2];
        args2[0] = jsdata;
        //round to 3
        int precision = 3;
        args2[1] = v8::Number::New(isolate, precision);
        v8::Local<v8::Value> args3[1];

        //call the javascript functions
        //measure CPU time with std::chrono::steady_clock
        c_start = std::chrono::steady_clock::now();
        v8::Local<v8::Value> averageJSResult = average-
Func->Call(context, global, 1, args1).ToLocalChecked();
        c_end = std::chrono::steady_clock::now();
        elapsed = c_end - c_start;
        auto averageJSTime = elapsed.count();

        c_start = std::chrono::steady_clock::now();
        v8::Local<v8::Value> sinJSResult = sin-
Func->Call(context, global, 1, args1).ToLocalChecked();
        c_end = std::chrono::steady_clock::now();
        elapsed = c_end - c_start;
        auto sinJSTime = elapsed.count();

        c_start = std::chrono::steady_clock::now();
        v8::Local<v8::Value> roundJSResult = round-
Func->Call(context, global, 2, args2).ToLocalChecked();
        c_end = std::chrono::steady_clock::now();
        elapsed = c_end - c_start;
        auto roundJSTime = elapsed.count();

        c_start = std::chrono::steady_clock::now();
        v8::Local<v8::Value> emptyFuncCallJSResult = empty-
Func->Call(context, global, 0, args3).ToLocalChecked();
        c_end = std::chrono::steady_clock::now();
        elapsed = c_end - c_start;
        auto emptyFuncCallJSTime = elapsed.count();

        c_start = std::chrono::steady_clock::now();
        v8::Local<v8::Value> firstItemJSResult = firstItem-
Func->Call(context, global, 1, args1).ToLocalChecked();
```

```
        c_end = std::chrono::steady_clock::now();
        elapsed = c_end - c_start;
        auto firstItemJSTime = elapsed.count();

        //call the C++ functions
        //measure CPU time with std::chrono::steady_clock
        c_start = std::chrono::steady_clock::now();
        double averageCPPResult = average(data);
        c_end = std::chrono::steady_clock::now();
        elapsed = c_end - c_start;
        auto averageCPPTime = elapsed.count();

        c_start = std::chrono::steady_clock::now();
        std::vector<double> sinCPPResult = calcSin(data);
        c_end = std::chrono::steady_clock::now();
        elapsed = c_end - c_start;
        auto sinCPPTime = elapsed.count();

        c_start = std::chrono::steady_clock::now();
        std::vector<double> roundCPPResult = cal-
cRound(data, precision);
        c_end = std::chrono::steady_clock::now();
        elapsed = c_end - c_start;
        auto roundCPPTime = elapsed.count();

        c_start = std::chrono::steady_clock::now();
        emptyFunctionCall();
        c_end = std::chrono::steady_clock::now();
        elapsed = c_end - c_start;
        auto emptyFuncCallCPPTime = elapsed.count();

        c_start = std::chrono::steady_clock::now();
        double firstItemCPPResult = firstItem(data);
        c_end = std::chrono::steady_clock::now();
        elapsed = c_end - c_start;
        auto firstItemCPPTime = elapsed.count();

        std::cout << "V8 startup time: " << V8StartupTime <
< " ms" << std::endl;
        std::cout << "In-
put data length: " << data.size() << std::endl;
        std::cout << "Javascript times" << std::endl;
        std::cout << "Average: " << aver-
ageJSTime << " ms" << std::endl;
        std::cout << "Sin: " << sinJSTime << " ms" << std::
endl;
        std::cout << "Round: " << roundJSTime << " ms" << s
td::endl;
        std::cout << "Empty function call: " << emptyFunc-
CallJSTime << " ms" << std::endl;
        std::cout << "First item: " << first-
ItemJSTime << " ms" << std::endl;
        std::cout << "C++ times" << std::endl;
```

```
        std::cout << "Average: " << averageCPP-
Time << " ms" << std::endl;
        std::cout << "Sin: " << sinCPP-
Time << " ms" << std::endl;
        std::cout << "Round: " << roundCPP-
Time << " ms" << std::endl;
        std::cout << "Empty function call: " << emptyFunc-
CallCPPTime << " ms" << std::endl;
        std::cout << "First item: " << firstItemCPP-
Time << " ms" << std::endl;

    }

    isolate->Dispose();
    v8::V8::Dispose();
    v8::V8::ShutdownPlatform();
    delete createParams.array_buffer_allocator;

    return 0;
}
```