INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA

isep

Mixed Reality (XR) Distributed Applications on the Web

NUNO BASTOS LIMA
Setembro de 2024

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Mixed Reality (XR) Distributed Applications on the Web

Nuno Lima

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Advisor: Dr. Nuno Pereira

Porto, September 15, 2024

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, September 15, 2024

Abstract

As Mixed Reality (XR) applications evolve and become more complex, devices show computing and power limitations. The Augmented Reality Edge Networking Architecture (ARENA) is a platform that simplifies the development and hosting of XR applications on the Web. ARENA allows applications to be executed in other remote hosts, which in turn provides a solution to distribute the execution of XR applications that are compiled to WebAssembly and make it safe by using sandboxed environments in specific runtimes. However, it is limited to centralized hosts and therefore cannot achieve a truly distributed system. Furthermore, focusing the computation power in centralized servers creates latency issues to mixed reality applications that should provide a fluent experience to users. This document describes the design and implementation of a solution to bring computation to devices on the edge of the network and to increase the scope of runtimes that could be used to execute the applications. The result is a browser-based runtime that communicates via the MQTT messaging protocol to execute WebAssembly modules remotely, supports multiple WebAssembly runtime engines such as Wasmer and Runno and has a management user interface to visualize and interact with module information. The process involved the research about WebAssembly and runtimes that could be integrated in the solution, analysis of the functional and non-functional requirements and design of the system including its architecture, deployment structure and processes. There was also a benchmarking process that revealed a solution with fast performance times and reliability, proved by averages of module execution under 500 milliseconds and sandbox setup under 60 milliseconds.

Keywords: Mixed Reality, Edge Computing, WebAssembly, Runtime, Application Distribution

Resumo

À medida que as aplicações de realidade mista (XR) evoluem e se tornam cada vez mais complexas, os dispositivos revelam limitações a nível de computação e energia. A Augmented Reality Edge Networking Architecture (ARENA) é uma plataforma que simplifica o desenvolvimento e alojamento de aplicações de XR na Web. A ARENA permite a execução de aplicações em *hosts* remotos, o que promove uma solução de distribuição de execução de aplicações de XR que são compiladas para WebAssembly e o faz de forma segura utilizando ambientes seguros nos seus runtimes. No entanto, estes são centralizados e limitados, o que impede a existência de um sistema verdadeiramente distribuído. Além disso, a centralização do poder de computação em servidores centralizados cria problemas de latência para aplicações de realidade mista que deviam oferecer uma experiência fluída aos utilizadores. Este documento descreve o desenho e implementação de uma solução para trazer a computação para dispositivos da rede e aumentar a abragência de runtimes que podem ser utilizados para executar as aplicações. O resultado é uma aplicação web para browsers que comunica através do procotolo de mensagens MQTT para executar módulos WebAssembly remotamente, suporta vários runtimes WebAssembly tais como Wasmer e Runno, e tem uma interface gráfica para visualizar e interagir com a informação dos módulos. O processo envolveu a pesquisa sobre WebAssembly e runtimes que pudessem ser integrados na solução, a análise dos requisitos funcionais e não funcionais, e o desenho do sistema incluindo a sua arquitetura, estrutura de implantação e processos. Houve também um processo de avaliação de performance, que revelou uma solução com tempos de execução rápidos e confiáveis, que é provado por médias de tempo de execução de módulos abaixo dos 500 milisegundos e criação de ambientes abaixo dos 60 milisegundos.

Acknowledgement

Finishing a master's degree is not an easy task to complete. It requires not only a giant amount of dedication, but also a lot of people's help. Therefore, there are some acknowledgements to be given.

To my family, for their unwavering support, trust, and patience throughout this stage of my academic journey.

To my close friends and colleagues, for their invaluable advice and the cherished memories we created during this period.

A special acknowledgment to Professor Dr. Nuno Pereira, my supervisor, for trusting me as his student and providing invaluable guidance throughout the development of this project and dissertation.

To Instituto Superior de Engenharia do Porto, for the extensive knowledge and opportunities provided from the first day of classes to the completion of my master's course.

Contents

List of Figures xiii

List of Tables xv

List of Source Code xvii

List of Acronyms xix

1 Introduction 1
- 1.1 Problem Definition and Motivation . . . 1
- 1.2 Goals . . . 2
- 1.3 Methodology . . . 3
- 1.4 Document Structure . . . 3

2 State of the Art 5
- 2.1 Edge Computing . . . 5
- 2.2 Mixed Reality . . . 7
  - 2.2.1 Definition and virtuality continuum . . . 7
  - 2.2.2 Use Cases and Examples . . . 7
  - 2.2.3 Mixed Reality Application Development . . . 9
- 2.3 Augmented Reality Edge Networking Architecture (ARENA) . . . 9
  - 2.3.1 Arquitecture Overview . . . 9
  - 2.3.2 ARENA Scenes . . . 10
  - 2.3.3 ARENA Applications . . . 12
  - 2.3.4 ARENA Runtime Supervisor (ARTS) . . . 13
- 2.4 WebAssembly . . . 14
  - 2.4.1 Historical Context and Motivation . . . 14
  - 2.4.2 Definition, Compilation and Execution . . . 14
  - 2.4.3 WASI . . . 16
- 2.5 WebAssembly Runtimes . . . 17
  - 2.5.1 Wasmer . . . 17
  - 2.5.2 Runno . . . 18
  - 2.5.3 Wasmtime . . . 18
  - 2.5.4 SpiderMonkey and V8 . . . 19
  - 2.5.5 Runtime Comparison . . . 19
  - 2.5.6 ARENA's WebAssembly Runtime . . . 19

3 Design 21
- 3.1 Requirements . . . 21
  - 3.1.1 Functional Requirements . . . 21
    - UC1: Register Runtime . . . 22

UC2: Start Module 22
UC3: Check Runtime Health 22
3.1.2 Non-Functional Requirements 22
3.2 General Concepts and Design Choices 23
3.2.1 Deployment Structure and Architecture 23
3.2.2 Communication 25
3.2.3 Messages 26
3.3 Processes 27
3.3.1 UC1: Register Runtime 27
3.3.2 UC2: Start Module 29
3.3.3 UC3: Check Runtime Health 31
3.4 Implementation 31
3.4.1 Module execution 32
Wasmer implementation 32
Runno implementation 33
3.4.2 Runtime User Interface 33

4 Benchmarking **35**
4.1 Methodology 35
4.2 Runtime Registration Benchmarking 35
4.3 Module Request Message Processing Benchmarking 36
4.4 Module Execution Benchmarking - Box 37
4.4.1 Module Setup 37
4.4.2 Module Execution 38
4.5 Module Execution Benchmarking - Authentication 39
4.5.1 Module Setup 39
4.5.2 Module Execution 39
4.6 Module Execution Benchmarking - AEAD 40
4.6.1 Module Setup 40
4.6.2 Module Execution 41

5 Conclusion **43**

Bibliography **45**

List of Figures

2.1 Edge Computing device ecosystem 6
2.2 Visual representation of the Virtuality Continuum 7
2.3 Use case of HoloAnatomy [18] 8
2.4 Augmented Reality application Pokemón GO [22] 8
2.5 ARENA's Architecture Overview 10
2.6 ARENA's Scene Editor 11
2.7 ARENA object definition message example 11
2.8 ARENA Scene loading and object real-time updating flow 12
2.9 WebAssembly (WASM) [34] 14
2.10 Compilation of C code to WebAssembly and JavaScript using Emscripten [32] 15
2.11 Portability of WebAssembly for different systems [41] 16
2.12 *putc* interface implementation possibilities [42] 16
2.13 Visual representation of WASI's portability [42] 17
2.14 Visual representation of Wasmer's WASI extension, WASIX [44] 18

3.1 Use case diagram. 21
3.2 Deployment diagram example for ARENA implementation 24
3.3 Component diagram representing the runtime architecture 25
3.4 Messaging standard as an inheritance relationship 27
3.5 Process Diagram for UC1 (register request) 28
3.6 Process Diagram for UC1 (handle manager response) 29
3.7 Process Diagram for UC2 (start module) 30
3.8 Process Diagram for UC3 (check runtime health) 31
3.9 Runtime user interface with example modules 34

4.1 Cumulative distribution function for runtime registration benchmarks 36
4.2 Cumulative distribution function for message processing benchmarks 37
4.3 Cumulative distribution functions for the box module setup times 38
4.4 Cumulative distribution functions for the box module execution times . . . 38
4.5 Cumulative distribution functions for the authentication module setup times 39
4.6 Cumulative distribution functions for the authentication module execution times 40
4.7 Cumulative distribution functions for the AEAD module setup times 41
4.8 Cumulative distribution functions for the AEAD module execution times . . 42

List of Tables

2.1 Program Object Definition 13
2.2 WebAssembly runtime feature comparison 19
2.3 Channel Definition 20

3.1 Register runtime use case specification 22
3.2 Start module use case specification 22
3.3 Check runtime health use case specification 22

List of Source Code

3.1 Runtime Configuration Structure in JSON 25

List of Acronyms

AR	Augmented Reality.
ARENA	Augmented Reality Edge Networking Architecture.
ARTS	ARENA Runtime Supervisor.
DNS	Domain Name Service.
IoT	Internet-of-Things.
MRTK	Mixed Reality Toolkit.
VR	Virtual Reality.
WASI	WebAssembly System Interface.
XR	Mixed Reality.

Chapter 1

Introduction

The introductory chapter serves as the starting point to understand the motivation and problem definition behind the project of this document. Furthermore, it lays down the main goals and objectives of this work, together with the methodology that was followed to achieve a solution and a section describing the structure of the document.

1.1 Problem Definition and Motivation

Mixed Reality (XR), described in more detail in chapter 2.2, is a technological concept that has seen an increase in popularity and impact in the industry, education and medical landscapes [1–3]. As the technology evolves and more XR applications are developed to perform higher resource intensive tasks, devices such as head-mounted displays and smartphones have significant computing and power limitations, which can influence the applications performance negatively.

With the evolution of these technologies and their complexity, XR applications need to perform multiple tasks to deliver an enhanced experience to the user such as odometry that keeps track of the position of the camera to render virtual objects, depth estimation to process distant information between objects and the camera, object detection tasks that identify physical objects in each camera frame with spatial awareness. XR applications require all these tasks and more to be executed at latency-critical performance and that leads to the possibility of offloading such computation power to cloud or edge servers [4].

The idea of offloading computation is already applied to multiple areas of work, such as gaming with solutions like NVIDIA GeForce Now or Xbox Cloud Gaming that allow players to play games without having to own expensive hardware by running the games in the cloud [5].

With mixed and augmented reality applications, offloading the computational tasks to other more capable devices and servers, such as the cloud, is an interesting option to alleviate some devices limitations. However, XR applications require low latency to provide the user with the most immersive experience and offloading computation to the cloud (Cloud Computing) would impact the performance and, therefore, hurt the user's immersion [6]. As an alternative, Edge Computing suggests the offload to be made to the edge of the network, where the applications would be closer to the data sources, consequently reducing latency and lead to faster response times [7].

The Augmented Reality Edge Networking Architecture (ARENA) is a platform that simplifies building and hosting collaborative XR applications on the Web [8]. Among its features, ARENA allows XR applications to be executed in different targets. However, to execute

distributed applications, ARENA is limited to centralized dedicated hosts and falls back to the performance problem when executing these applications due to the lack of proximity between the clients and the servers that execute such applications. In order to realize a truly distributed system, a solution that integrates into the ARENA framework to execute applications in the browser, as well as manage, distribute and deploy those applications must be designed and implemented.

The solution to be designed requires deep knowledge of the ARENA Framework and its runtime infrastructure, including how to dispatch and execute applications. Furthermore, there is a need to relate these technologies to broader concepts related to distributed systems management and security. There are various design choices that can be made, which result in a solution that is not obvious but can be generalized to reach a wider range of contexts, like the ability to easily implement new engines or a messaging standard to maintain coherence while communicating with external systems. The problem also involves a scale of smaller problems and factors to tackle like security to avoid damages of WebAssembly modules that could be created to arm the machine it is running on and performance not only in the module execution but also in the creation of the environments for such modules. This also involves the structuring and creation of multiple components that focus on issues such as the deployment, monitoring and management of applications.

This work will provide the infrastructure support for Augmented Reality (AR) applications and more, which hold promise for dramatically transforming the interaction with digital systems, with broad applications in manufacturing, education, medicine, and much more.

1.2 Goals

The primary goal of this project is to explore edge computing as a form of offload for mixed reality applications, specifically to develop a browser implementation of a runtime for WebAssembly modules capable of supporting the execution of distributed applications. This can be dissected into smaller and more detailed objectives:

1. **analyse the ARENA Runtime System and state-of-the-art concepts -** the ARENA Runtime System is complex and in order to design a proper solution, its architecture must be studied and examined. There is also a plethora of technologies that can be used to implement the solution. An analysis of such technologies must be done, as the choices will affect the design of the system

2. **design and implement a runtime environment in the browser -** after studying the concepts related to the problem, the solution must be designed and implemented as a runtime environment in web browsers that complements the ARENA framework, in order to allow distribution of applications to edge devices closer to the users;

3. **design and implement a monitoring and management console -** there is also need to implement a console for monitoring and management of the applications being deployed and executed in the runtime environment in the browser;

4. **evaluate the solution -** when implemented, the system will be tested and evaluated, namely the performance of the runtime at the start and execution of applications. A benchmarking process must take place in order to evaluate the main processes of the runtime, specially the application execution that must be tested with multiple examples and instances.

1.3 Methodology

The development of this project followed a methodology that allowed for an investigation about WebAssembly runtimes as a way to achieve computation offloading, as well as the design of a solution that would work for multiple systems.

1. **Research about WebAssembly runtimes** - the gathering of different solutions for executing WebAssembly modules, comparing all features between them and possible browser support;
2. **Planning and Designing a solution** - with all the requirements that needed to be met, a solution was designed from the ground up including it's own architecture, communication protocols, message standards and other key aspects;
3. **Implementation phase** - based on the output from the design process, the components were implemented together with the use of the chosen WebAssembly runtimes;
4. **Performance analysis and benchmarking** - after the browser-based runtime was implemented, a series of tests and benchmarking analysis were done to evaluate the performance of the solution, as well as it's consistency.

1.4 Document Structure

This document is organized into a series of chapters, where each one addresses the aspects of the methodology used to develop this project.

The current introduction chapter aims to explain the problem and motivation behind the need of a system that allows module execution to be distributed on the web, as well as describing the goals of this project and methodology used in order to achieve the solution.

A state of the art chapter follows as a way to explore all the different concepts that the project involves like edge computing, computation offloading, mixed reality applications and also WebAssembly characteristics and runtimes.

After describing the state of the art concepts, the design chapter dissects the requirements of the project dividing them in use cases, explains the general concepts, architecture and design choices relative to the deployment structure of the runtime and defines the communication protocols and messaging standards used to bridge the runtime to the manager. An implementation section goes through some of the development aspects about module execution, integration with WebAssembly runtimes and the management console user interface.

The benchmarking chapter is a way to analyze the performance of the built solution, not only how fast it is while executing modules but also other processes like runtime registration and message processing. It makes use of data gathered from multiple tests and explores it using cumulative distribution functions.

The document finishes with the conclusion chapter that summarizes what was written in the body, from implications of the results explained during benchmarking, recommendations for future work around the project and what was achieved in the end.

Chapter 2

State of the Art

This chapter aims to describe and analyze the current technological landscape relevant to the key themes and tools associated with this project. It starts by exploring edge computing as an alternative to offload computation, followed by mixed reality concepts and applications, the Augmented Reality Edge Networking Architecture (ARENA) and then delves into the WebAssembly language and WebAssembly runtimes.

2.1 Edge Computing

The evolution of the Internet-of-Things (IoT) devices and network capabilities has led to new, more advanced applications that require low latency and data privacy. Most IoT applications are built around a centralized infrastructure where data processing and computing is done in the cloud, originating the term Cloud Computing. However, Cloud Computing is not viable in some use cases because of its high latency, difficulty posing to privacy and, in some cases, the high volume of generated data by end-devices, which can cause congestion, bandwidth waste and low performance. [7]

Edge Computing, the concept of bringing the computation closer to the data sources, was conceived to enhance performance in IoT applications that require low end-to-end latency by placing computing resources (cloudlets) at the edge of the network [9].

Figure 2.1 represents the edge computing device ecosystem, with end devices being in the outer part of the circle, the cloud in the center and between them is the edge with distributed cloudlets.

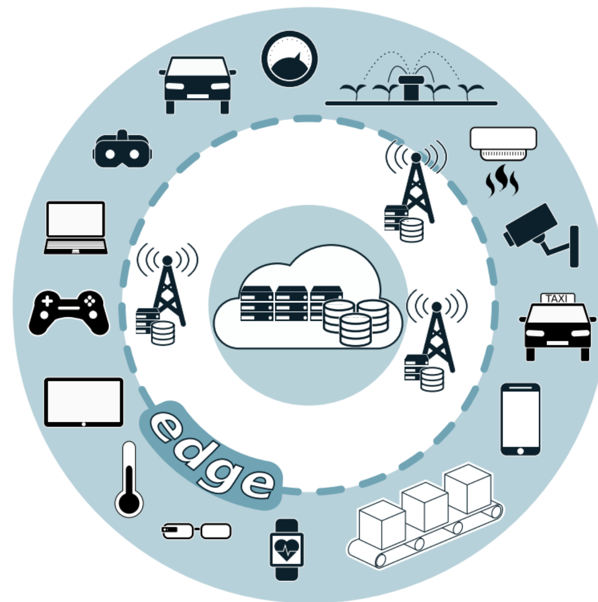

Figure 2.1: Edge Computing device ecosystem [10]

The proximity of cloudlets to end devices is advantageous in several ways compared to Cloud Computing [11]:

- **Responsiveness:** as the computing resource is physically closer to the mobile device it is easier to achieve low latency and high bandwidth;
- **Scalability:** high data rate sensors in IoT like video cameras in a security system generate a lot of data to be sent to the cloud constantly. With cloudlets nearby, the footage is analyzed in the edge of the network and only the results are then sent, reducing ingress bandwidth to the cloud;
- **Data privacy:** by serving as the first point of contact in the system, cloudlets can handle the processing of critical private data locally prior to the release of data into the cloud.

Edge computing is employed across various scenarios and use cases, but really shines in applications that require real-time processing. In healthcare systems edge computing can be used to analyze medical data in real-time. Smart home devices, such as thermostats, can use edge computing to reduce response times in cases of emergencies. In industry settings it can be utilized to prevent dangerous incidents in workers' environments. Even in the automotive sector edge computing allows an increase in data processing speed to help cars avoid obstacles while driving [12] [13] [14].

Computation offloading is the process of running an application, or parts of an application, outside the client device. It is motivated by insufficient computing resources or resources that could produce inaccurate results. Edge computing enables computation offloading by making resources available as cloudlets to handle critical tasks or any application execution that is sent by edge devices [10].

2.2 Mixed Reality

2.2.1 Definition and virtuality continuum

Mixed Reality (XR) is a technology that blends elements from Augmented Reality (AR) and Virtual Reality (VR) to create environments in which digital and physical objects coexist and interact in real-time. AR refers to the integration of 3D virtual objects into the real life environment, by overlaying such objects on top of the physical real world view. When using AR capable devices the user sees the world that is in front of him as it is with digital objects overlayed on top of it, while in VR the user is totally immersed on the full digital experience. VR is most often achieved with opaque headsets that totally block the vision of the person wearing it [15].

In 1994, Paul Ingram and Fumio Kishino introduced the "virtuality continuum", a concept that defines certain classes of these technologies and situates them in a spectrum. In this spectrum, represented in Figure 2.2 (adapted from the original paper), the real world environments are placed in the left end, while in the opposite end are located the virtual environments. AR, as a mixture of the real world with some digital elements, is placed near the real world boundary of the spectrum, while VR is located at the right end being the full virtual environment itself. Mixed Reality is also represented in the spectrum, covering the space between augmented reality and virtual reality because, as mentioned earlier, is a mixture of both [16].

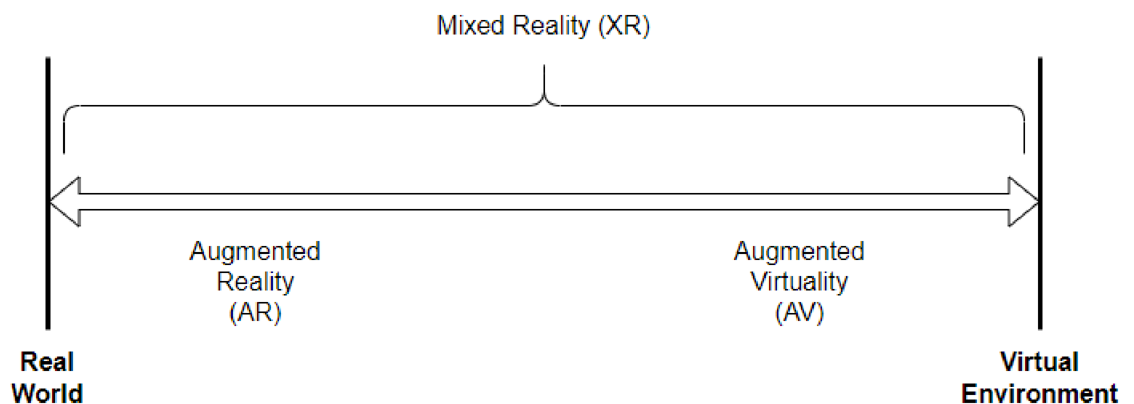

Figure 2.2: Visual representation of the Virtuality Continuum [16]

2.2.2 Use Cases and Examples

There are several examples of mixed reality applications being used in various sectors of the industry. From education to entertainment, this section covers some use cases of this technology applied to different scenarios.

- **Education** - Mixed Reality provides immersive settings that enhances motivation and learning [17]. There are a lot of mixed reality applications developed to be used in classrooms and research that can serve as examples of the evolution of these technologies in this field, such as HoloAnatomy [18] or zSpace [19];

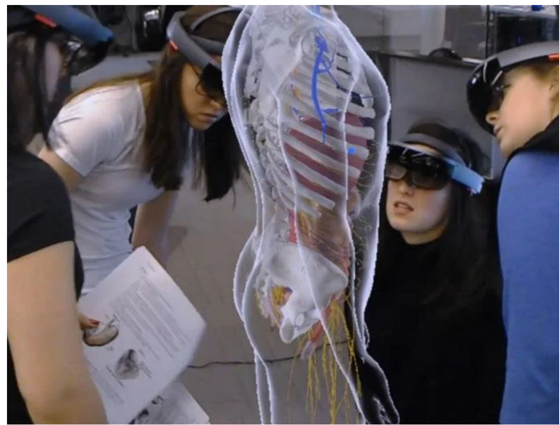

Figure 2.3: Use case of HoloAnatomy [18]

- **Healthcare** - Mixed Reality has also emerged as a valuable resource in healthcare, with systems used for medical surgery training or enhancements to virtual telemedicine between doctors and patients by creating an actual 3D depiction of a person [20].
- **Manufacturing** - the potential of mixed reality has been demonstrated in factories, with systems that allow manufacturing instructions to be displayed to workers without taking the look from assemblies as examples [21].
- **Video games and Entertainment** - plenty of video games have been developed to provide immersive environments that take users through all kinds of experiences. Popular games such as "Pokémon GO" (figure 2.4), "VR Chat" or "Half-Life: Alyx" are examples of such use of mixed, augmented and virtual reality in the entertainment industry.

Figure 2.4: Augmented Reality application Pokemón GO [22]

2.2.3 Mixed Reality Application Development

While the development of mixed reality applications faces challenges like real-time environment mapping, 3D model placement, sensoring, calibration and latency, there are plenty of tools that help developers build applications that produce immersive experiences. Such tools are based in various languages and have different characteristics. Unity is a popular game development engine that also supports augmented reality application development [23], ARKit and ARCore focus on building AR experiences for iOS and Android devices respectively [24] [25] and Mixed Reality Toolkit (MRTK) provides different tools integrated with Unity [26].

The development of mixed reality applications can also be taken to the web. AR.js is a JavaScript library that includes features like image tracking, location-based AR and marker tracking [27]. A-Frame is a web framework for building virtual reality experiences that is based in a entity-component framework [28]. three.js is a library that allows developers use 3D models on web applications [29]. These are several examples that help developers build web applications. However, none of them let users actually build their own scenes in a browser like the Augmented Reality Edge Networking Architecture (ARENA).

2.3 Augmented Reality Edge Networking Architecture (ARENA)

A project led by Carnegie Mellon University's Wiselab in collaboration with other universities such as Berkeley, UCLA and UCSD [30], the Augmented Reality Edge Networking Architecture (ARENA) is a platform that simplifies the development and deployment cycles of collaborative extended reality applications. It allows developers to easily host user-interactive applications in a 3D environment and scale collaborative multi-user XR applications. [8]

ARENA provides several components such as a hierarchical geospatial directory service to connect users to nearby content, a token-based authentication system to control user access and a runtime supervisor to dispatch programs across any network connected device. It uses modern XR Web technologies to extend compatibility through a plethora of different devices like tablets, smartphones, headsets and desktop browsers. [8]

Unlike tools such as ARKit [24], ARCore [25] and 3D game development platforms like Unity [23], ARENA provides a unified solution that instead of neglecting collaboration between remote users, offers networking interconnect with important services for augmented reality application development like geo-referencing content and access control. [8]

The project organises itself to achieve its main goal: scalability. The content in ARENA is structured with *Scenes*, an abstraction that contain a set of virtual assets like 3D objects, parameters and user interaction applications. A group of *Scenes* forms a *Realm*, which hosts services like web servers, a Publish-Subscribe messaging bus and a resource manager [8].

In this chapter, ARENA's architecture and main components (such as *Scenes*, *Realms*, message bus, application runtime and other services) will be explained in detail to construct the groundwork for the project documented in this dissertation.

2.3.1 Arquitecture Overview

ARENA's architecture is represented in figure 2.5. As mentioned in section 2.3, ARENA has a geospatial directory service called ATLAS that was built to allow users to find nearby content based on their location and manage the necessary data to link the Scene content with the physical world. [8]

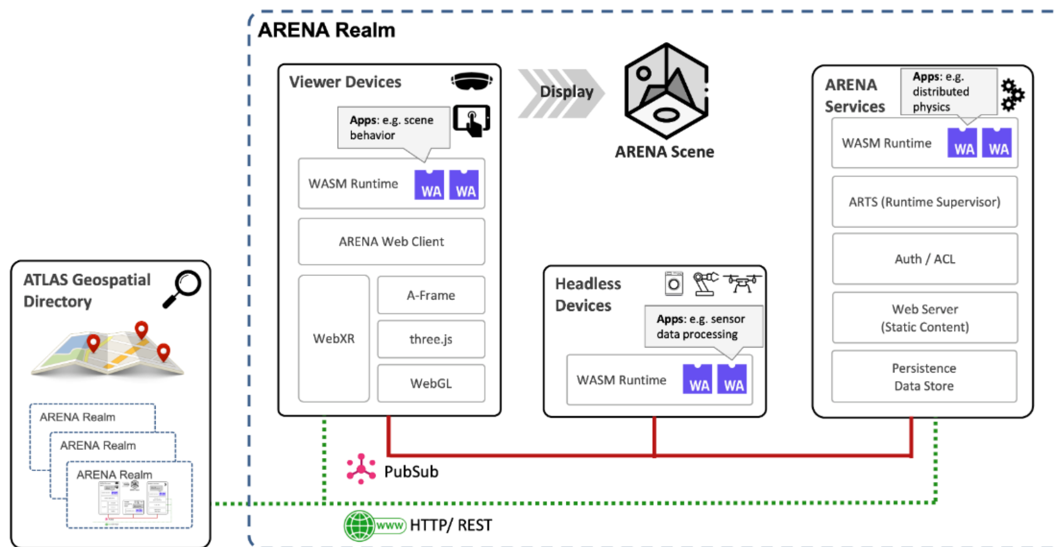

Figure 2.5: ARENA's Architecture Overview. [30]

The ATLAS service is used to register and discover ARENA Scenes and simplifies the process of combining multiple tracking technologies into a uniform coordinate system. ATLAS works the same way as the Internet's Domain Name Service (DNS), but instead of using domain names, it uses a mix of GPS coordinates and UUID markers to identify and locate Scenes. [30]

A Realm is a server that hosts ARENAs 3D content and services. It connects hardware components like viewing devices (headsets, smartphones or tablets) to other headless devices embedded into the environment (like cameras or sensors used for localization). Realms also have a set of services that expose REST APIs to query states, permission or create access tokens for users [30]. Some of these services are described below, but the Runtime Supervisor is explained in more detail in section 2.3.4. [8]

- **Persistent Data Store** - previously mentioned in section 2.3.2, its role in the ARENA system architecture is to provide all 3D objects within a scene and keep track of the latest state of any persistent object.
- **Web Server** - this component serves the purpose of storing static content like 3D models, images, sounds or even basic Scene skeletons.

2.3.2 ARENA Scenes

A Scene is defined as an abstraction that contains a group of related virtual assets such as 3D objects, configuration parameters and applications. [30]

Scenes are usually attached to a physical location and their access control is configurable, like a web application with a URL endpoint in a web server. However in most web browsers, a user can only access a website one at a time per tab, but in ARENA it is possible for a user to see and interact with objects from multiple Scenes in a public space, given that the user has access to the Scenes that contain such objects and applications [30].

Through a Web interface, developers can make changes to their Scenes and visualize its properties and objects. Figure 2.6 shows an example of a Scene's object list that includes options, a program, lights and GLTF models visualized in the Scene Editor [8].

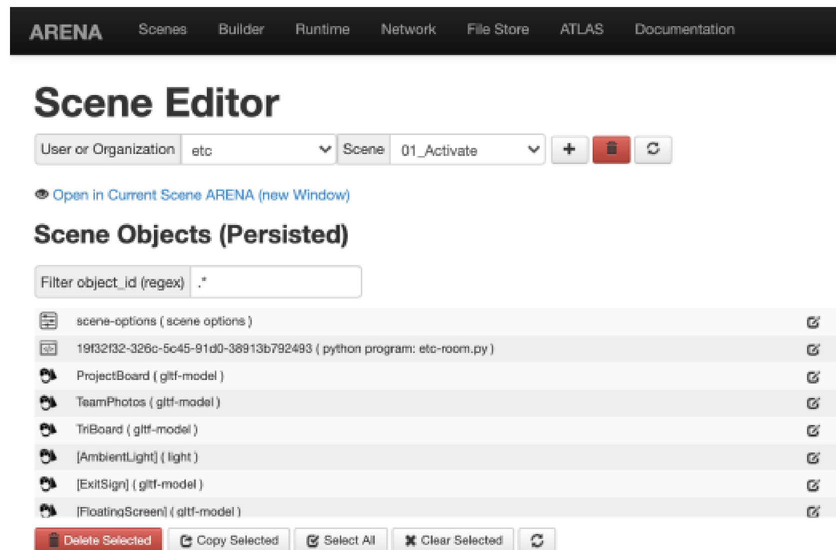

Figure 2.6: Scene object list visualized in the Scene Editor [8]

Using A-Frame's Entity Component-System architecture [28], an ARENA Scene can also be considered a collection of entities to which components can be attached. As ARENA follows A-Frame's primitives, it also adds specific components for AR markers, programs and networked events. Every object is based off a schema that defines its structure and the basis for the messages that are transmitted through the message bus. These messages have an identifier, *type* and an *action* (create, update or delete). An example of such message can be visualized in figure 2.7, which represents the creation of a box geometry object with depth, height, width, position and rotation values, as well as an ARENA-specific AR Marker. [8]

```
{ "object_id": "abox",
  "persist": true,
  "type": "object",
  "action": "create",
  "data": {
    "object_type": "box",
    "depth": 1,
    "height": 1,
    "width": 1,
    "position": { "x": 1, "y": 1, "z": 1},
    "rotation": { "x": 0, "y": 0, "z": 0},
    "armarker": {
      "lat": 40.4432,
      "lon": 79.9428,
      "markerid": "1",
      "markertype": "apriltag_36h11",
      "size": 150,
      "ele": 200
    }
  } }
```

Figure 2.7: ARENA object definition message example. [8]

Scenes are loaded the same way as web applications are loaded in a web browser, but as mentioned earlier in this section, ARENA allows the possibility of loading multiple scenes without switching between tabs [8]. Figure 2.8 describes the flow of Scene loading and object real-time updating. Once a user loads a particular Scene, the browser is given the 3D objects of such Scene that are stored in a Persistence Data Store that tracks the latest state of those objects and when the objects are fully loaded each one can be updated through the message bus. [30]

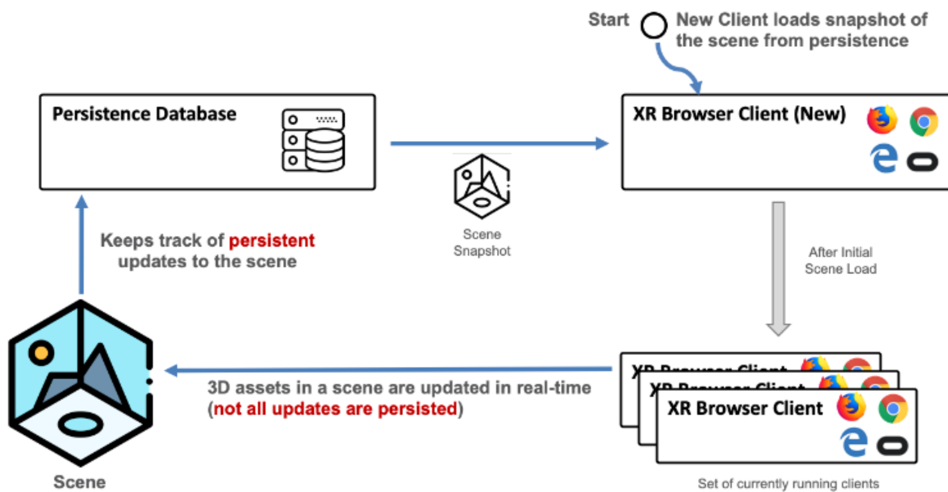

Figure 2.8: Scene loading and object real-time updating flow. [30]

The access control is managed with a token-based strucure at Scene level. Each user is given a token that defines the read/write access that he has in a Scene. If a user does not have read permissions for a Scene, all objects of such Scene will be invisible for that user. [30]

2.3.3 ARENA Applications

Tight interactions between the several system components (e.g. devices carried by users, outside-in sensing for tracking and localization), make it hard to support scalable XR applications using traditional tiered cloud-edge solutions.

The ARENA architecture was designed specifically to support latency-sensitive local interactions at the edge, while still taking advantage of the traditional cloud infrastructure. ARENA leverages this architectural feature further by defining a framework for the development of applications. The framework is inspired in the Actor Model [31] and the main idea is to allow state-full entities (programs) to specialize on managing a particular resource or virtual object and let them communicate using message passing. It is similar to approaches in many professional activities. For example, during a surgery, each member of the medical team has a very specialized task, and is often responsible for an instrument/resource/area that no other member in the team operates directly. Obviously, members communicate among each other and often ask others to carry out actions.

2.3.4 ARENA Runtime Supervisor (ARTS)

By leveraging the common runtime and resource monitoring integrated into the message bus, the ARENA Runtime Supervisor (ARTS) manages the heterogeneous compute resources of an ARENA Realm. Besides 3D objects, ARENA Scenes include the program objects that specify how a program is started. As a user loads an ARENA scene, program objects originate requests to ARTS, that, in turn, will forward these requests to one of the available runtimes that previously registered with ARTS.

Program Instantiation: A program object, besides information about where to find the program files and start them (entry file, arguments, environment variables), also indicates how the program is instantiated. Programs can be instantiated for each client device, for example for programs that are dedicated to manage some resource or service for a particular user device, or per ARENA scene, meaning that the program is maintaining a global state for that scene. There are also plans to support background service programs that are instantiated globally. Program objects in a scene can also indicate their affinity, i.e. which runtime the program should run. This is interpreted by ARTS as an indication, which it may or may not follow.

Program Instance Lifetime: The instantiation mode of a program defines its lifetime. Programs instantiate per client are terminated when the client exits, which is detected by a combination of browser session bookkeeping, and message bus features that allow to send a last-will messages when clients disconnect abruptly. Per scene programs have a more interesting semantics: they are started when the first client device loads a scene, and subsequent client devices will not be able to start another instance. The program is terminated when no activity is detected in an ARENA scene for a defined timeout. Activity in a scene in determined through a keep-alive mechanism performed between runtimes and ARTS. Table 2.1 represents the definition of a program object.

Table 2.1: Program Object Definition

Field	Description
name	Name of the program (programs have a know location based on a username namespace)
affinity	Indicates the module affinity (client=client's runtime; none or empty=any suitable/available runtime)
instantiate	Single instance of the program (=single), let every client create a program instance (=client), or a global service (=global)
filename	Filename of the entry binary. e.g. "arb.py" (required)
filetype	Type of the program (WA=WASM or PY=Python)
apis	APIs required by the module e.g.: [wasi:clock]
args	Command-line arguments.
env	Environment variables. Supports variables as above.
channels	Channels describe files representing access to IO from pubsub (possibly more in the future; currently only supported for WASM programs).

2.4 WebAssembly

This section describes WebAssembly, a technology that is involved in the goals of the project. Currently, ARENA uses WebAssembly to execute applications through its runtimes, by making use of WebAssembly sandboxes which provide security and portability. The section starts with the motivation and historical context behind the it's creation, practical use cases, main advantages and it's future. WebAssembly is what makes this project possible and is at the core of the developed solution.

2.4.1 Historical Context and Motivation

As web applications and technologies evolve through time and mature to more sophisticated and interactive experiences with 3D visualization, audio and video software and even video games, code efficiency and security gained a lot more importance. At the time WebAssembly was introduced, JavaScript was the only natively supported programming language on the Web and the compilation target for other languages, which especially in the latter case revealed inconsistent performance issues [32].

WebAssembly, first released in 2017 and created by developers of the four major browsers, targets the goals of fulfilling the properties of a low-level compilation target, as one should have its semantics be safe and fast to execute, indepedent from any language, hardware and platform, aswell as being compact and easy to validate, generate and compile. [33]

Figure 2.9: WebAssembly (WASM) [34]

Before WebAssembly, there were several attempts to bring low-level code to the web. Microsoft's ActiveX failed on achieving safety on its technical construction, Native Client [35] failed in the portability aspects as it was exclusive to Chrome and Emscripten [36] (at the time used to compile C and C++ code to asm.js [37]) was tightly connected to JavaScript's semantics, which would have been dependent on JavaScript's evolution to evolve itself, leading to it being inefficient. [33]

2.4.2 Definition, Compilation and Execution

WebAssembly is a binary instruction format that was designed to be ran at near native performance in web browser environments. Developers can write their source code from a plethora of programming languages like C or C++ and through compiler toolchains generate WebAssembly binary. [38]

WebAssembly binaries are generated through language dedicated compilers, such as Emscripten [36] or Low Level Virtual Machine (LLVM) [39]. Figure 2.10 shows a comparison between a simple "Hello World" program source code compiled to WebAssembly binaries and to JavaScript code.

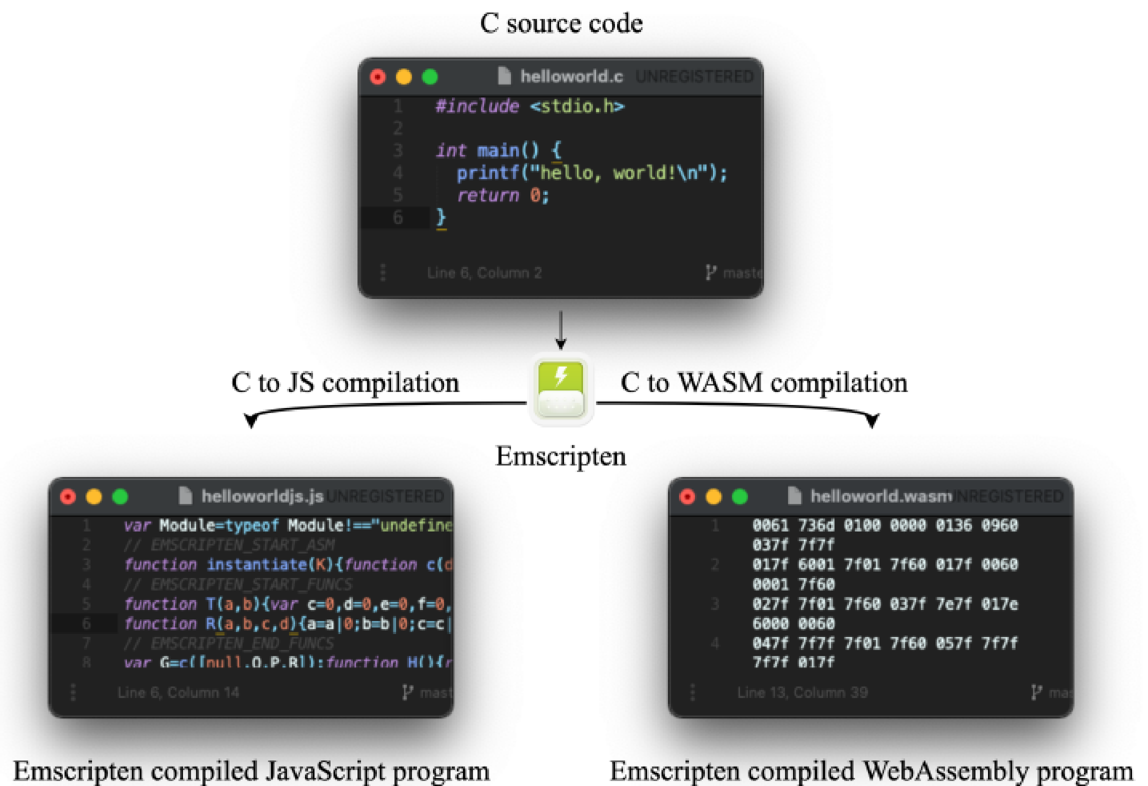

Figure 2.10: Compilation of C code to WebAssembly and JavaScript using Emscripten [32]

While Javascript and WebAssembly are not the same thing and are meant to work together, it is possible to compare the way their binaries are loaded and decoded in a browser environment. Javascript code, that is manually written by the developers, is decoded and parsed at runtime while WebAssembly programs are delivered as binaries, compiled from high-level languages, which leads to faster load times. [40]

After being compiled from the original source code language, a WebAssembly module (in the bytecode format) targets a stack-based virutal machine that can execute in different modes [41]:

- **Just-in-Time (JIT) compilation** - the code is compiled to native machine code as it is being executed, leading to a performance increase;
- **Ahead-of-Time (AoT) compilation** - the code is compiled ahead of the programs execution, leading to faster start up times but less flexibility.

WebAssembly was originally meant to run in web browsers, but researchers have been trying to take it beyond the web to Internet of Things (IoT), edge computing and other use cases, thanks to the portability and security that it offers. WebAssembly has a sandboxed linear memory and structured control flow that provide secure memory access, as well as it being agnostic to source languages, meaning that after being compiled to WebAssembly binaries, an application can be executed in different hardware architectures [38]. Figure 2.11 helps visualizing the portability of WebAssembly in different hardware systems.

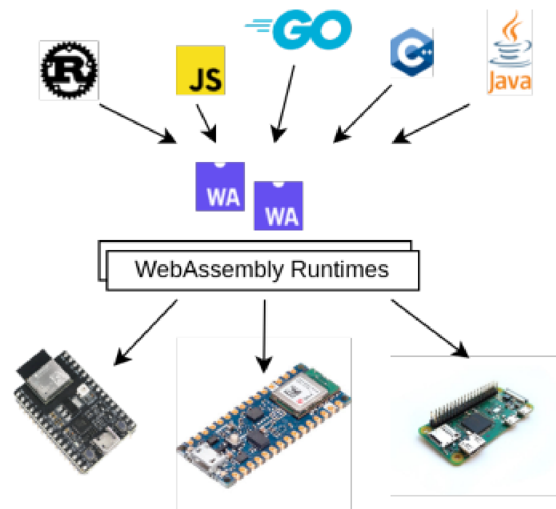

Figure 2.11: Portability of WebAssembly for different systems [41]

2.4.3 WASI

As described in the previous section, developers started to push WebAssembly and it's capabilities beyond the browser. However, WebAssembly is a language that was built to be executed in conceptual machines, therefore taking it to operating systems requires a system interface to support system calls across all platforms [42].

When developers write programs in any language, they don't need to worry about system calls or what system will the program be working on because most of the languages use abstraction. It's the compiler's responsibility to understand what system will it be targeting and choose the right implementation of the system's API. Figure 2.12 represents this premise with the example of the *putc* interface, which the compiler will have to choose between two implementations based on the machine it is targeting [42].

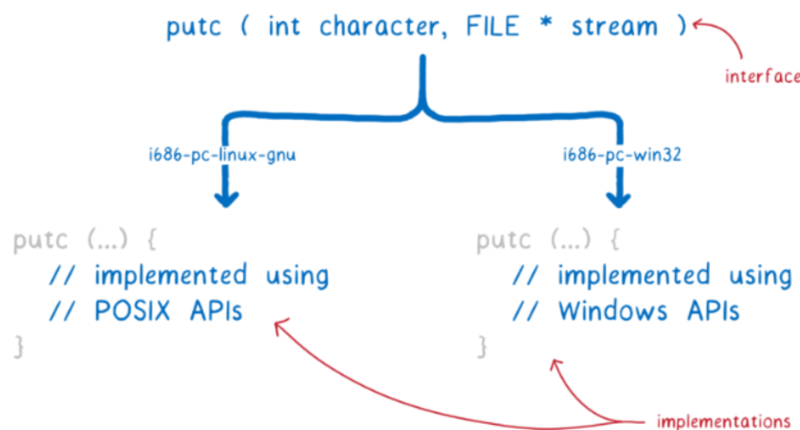

Figure 2.12: *putc* interface implementation possibilities [42]

However, WebAssembly needed to take this step even further due to its portability aspects.

When compiling a WebAssembly module, the targeting system is still unknown, so the compiler will not understand what system API to use. To address this issue, the WebAssembly System Interface (WASI), that creates a new layer of abstraction and releases the compilers responsibility of choosing the correct system API and takes it to the runtime where the application will be executed on. Figure 2.13 helps visualizing the process of compiling and executing a WebAssembly module, as the compiler targets WASI and the runtime where the program will be executed implements the system interface [42].

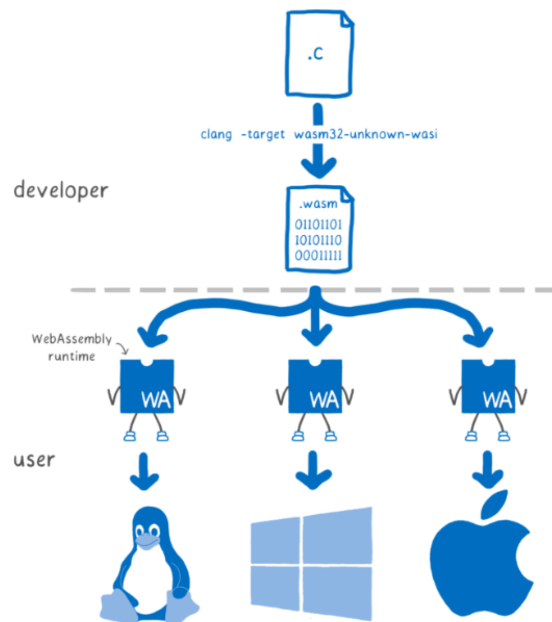

Figure 2.13: Visual representation of WASI's portability [42]

2.5 WebAssembly Runtimes

A WebAssembly runtime is a software application that executes WebAssembly modules. At the very core of the solution described in this document exists a runtime that will load the binary files and execute modules in a sandboxed environnment.

In this section, several WebAssembly runtimes will be listed and compared, in order to justify the use in the final solution. The choices for what runtime to use were based on the need for WASI implementation and browser support.

2.5.1 Wasmer

Wasmer is an ecosystem around WebAssembly that allows developers to execute WebAssembly modules anywhere like in the browser or embedded in another languages program. It involves a standalone runtime for all operating systems, a JavaScript SDK to execute modules in the browser and a concept of user published containers available in a public registry to be used by other developers [43].

As the runtime does implement WASI, Wasmer took an extra step in its approach by extending its features and creating WASIX, a superset of WASI with the goal to make WebAssembly modules more compatible with POSIX programs [44]. Figure 2.14 helps visualizing how WASIX not only encapsulates all of WASI, but also extends it.

Figure 2.14: Visual representation of Wasmer's WASI extension, WASIX [44]

The Wasmer JavaScript SDK is a library developed to execute WebAssembly modules in the browser and integrate them in web applications. Two different packages of the SDK were analyzed: the *@wasmer/wasi* is an older version that gives the developer the possibility of instantiating the modules using the core WebAssembly library, which in turn allows for JavaScript functions to be imported to the module sandbox; the *@wasmer/sdk* is the most recent iteration and supports WASIX features, but lacks the possibility of importing JavaScript functions to modules, which is the main reason *@wasmer/sdk* was picked in its favor [45].

2.5.2 Runno

Runno is a runtime that allows developers to execute code from languages like Python, Ruby and others in the browser by using WebAssembly and WASI. The runtime itself was not used in the final solution of the project, but Runno ships with *@runno/wasi*, the package that allows the runtime to execute WebAssembly modules [46].

As the package was built only for the web and has a WASI implementation, Runno is an available option as an engine for the final solution of the documented project, together with Wasmer.

2.5.3 Wasmtime

Bytecode Alliance is an organization that works on standardizing and building foundations for WebAssembly and WASI compilers and runtimes. The organization developed its own runtime called Wasmtime, that has supports a rich set of WASI's API [47].

Because Wasmtime is not a runtime for the browser, it was not used in the project, but it is worth to mention as in the future the project could evolve to being not only in the browser, but also for standalone servers.

2.5.4 SpiderMonkey and V8

SpiderMonkey and V8 are, respectively, Mozilla's and Google's JavaScript/WebAssembly engines for they're browsers. Both of them have a WebAssembly library that is the core for compiling and instantiating modules, as well as the foundation for most of the runtimes mentioned above.

The core WebAssembly library does not support nor implement WASI's API, but allows developers to import JavaScript functions into the modules. As matter of fact, the WASI implementations of the runtimes are imports sent to the modules in order to support calls.

Creating a WASI implementation was not in the plans for this project. However, the chosen libraries for this project complement the core WebAssembly library and it was used to instantiate modules.

2.5.5 Runtime Comparison

This section presents a summary of the previously described WebAssembly runtimes and a comparison between them. The main topics in the decision making process were the WASI and Web browser compatibility of such runtimes, because it is a focal point for the project to work in the intended truly distributed architecture. The comparison can be visualized through table 2.2.

Table 2.2: WebAssembly runtime feature comparison

Name	WASI Support	Browser Compatibility
Wasmer	Yes	Yes
Wasmtime	Yes	No
Runno	Yes	Yes
WebAssembly Javascript Library	No	Yes

After analyzing the core features, Wasmer and Runno were the chosen engines to execute WebAssembly modules in this project. The standard JavaScript library for WebAssembly is used to complement both runtimes.

2.5.6 ARENA's WebAssembly Runtime

ARENA also has it's own implementation of a WebAssembly runtime, which implements all necessary interfaces including WASI and specific ARENA features for it's modules.

WebAssembly runtimes can run on local edge devices, including edge servers, headsets and embedded computing platforms, or even in the cloud, and provide a common runtime functionality upon which ARENA applications can be built, and allows to execute and manage the life-cycle of ARENA applications. ARENA has implementations of common runtime for Linux and embedded platforms by leveraging the WASM micro runtime [48]. Runtimes use the pub/sub infrastructure to register their availability. When registering, they indicate some information about their resources, such as how many modules they can run, and the host APIs they support. Those runtimes implement WASI [49], and, for example, the APIs information indicate which WASI version, or possibly, as they become available, which WASI modules [42] are supported by the runtime, e.g.: `wasi:snapshot_preview1`,

`wasi:unstable, wasi:core, wasi:sensors`. This is also used to indicate that the runtime implements ARENA's own system interface APIs, for example, `awlib:channels` indicates that the runtime implements a channels API as described next (`awlib` stands for ARENA WASI library).

WASM modules are given access to channels, which, to the WASM module, look like files and provide access to networked resources such as sockets and the pub/sub hierarchy. Channels are capabilities defined as part of the parameters needed to start a module. Table 2.3 shows a channel definition. ARENA currently supports `pub/sub` channels (indicated in `type`) and plan to support `client` (for client socket connections). The `params` field includes some channel type specific parameters, such as the pub/sub topic that ARENA is giving access to, or host and port for client sockets. Each channel definition creates two message-oriented streams that are accessed as files: one with the pub/sub data (`/ch/<channel-name>/data`) and the other information about the channel (`/ch/<channel-name>/info`).

Table 2.3: Channel Definition

Field	**Description**
path	Channel path visible by the program.
type	Channel type. Can be pubsub, or, in the future, client socket.
mode	Access mode [read (r) , write (w), read/write (rw)]
params	Type (i.e. pubsub/client)-specific parameters.

The WASM runtime accepts requests to create/delete WASM modules. Requests to create modules indicate where the program files can be found, the entrypoint (how to start the program), APIs required by the module, environment and command line parameters to be passed, and, importantly, a definition of the channels the module has access to. The information about APIs required by the module is used to select eligible runtimes by the *runtime supervisor*, which is responsible for managing the life-cycle of WASM runtimes and WASM modules. The runtime supervisor is presented in more detail in section 2.3.4.

There are implementations of the ARENA runtime that run on Linux, and a prototype browser runtime will be the subject of this work.

Chapter 3

Design

This chapter presents the solution's design choices and architecture. It goes through functional requirements in the form of use cases and uses the FURPS model to list non-functional requirements that define what the application needs to become reliable. Then, the architecture and deployment structure of the solution is detailed, together with some key concepts like messaging and communication, process design and implementation aspects regarding the execution of modules.

3.1 Requirements

This section is dedicated to the analysis of the requirements that the solution must satisfy in order to function. These requirements are divided in two major categories: functional and non-functional.

3.1.1 Functional Requirements

The functional requirements of the project are related to the life cycle of the runtime itself, as well as the execution of modules. Figure 3.1 is a use case diagram that displays the main functional requirements of the project.

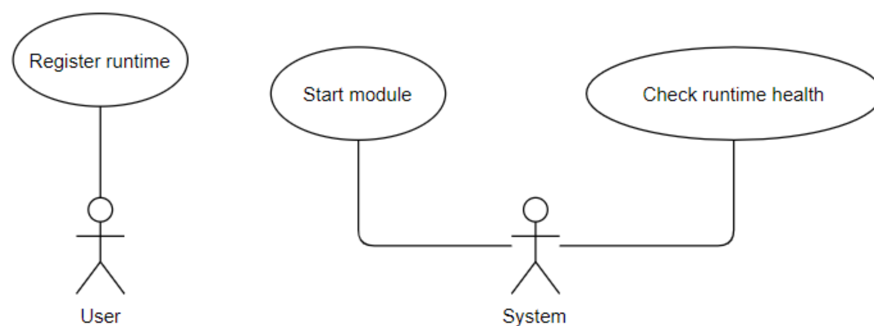

Figure 3.1: Use case diagram.

Below, each functional requirement is individually described with acceptance criteria, inputs and respective dependencies.

UC1: Register Runtime

This use case occurs when the user wants to initialize the runtime and register the machine as available to execute modules. The registration of the runtime serves the purpose of letting the manager know that the instance exists and save the data needed to start modules.

Table 3.1: Register runtime use case specification

Parameter	Value
Acceptance Criteria	The runtime state should change to "Listening to requests"
Input	MQTT Broker data
Output	None
Dependencies	The MQTT Broker must be running.

UC2: Start Module

This use case is the focal point of the project. The runtime must execute WebAssembly modules in the browser, triggered by a system that needs it. The way the runtime receives the module information and starts the module execution will be described further in the document.

Table 3.2: Start module use case specification

Parameter	Value
Acceptance Criteria	The runtime starts the execution of the module and notify the system
Input	Module data
Output	Module data notification
Dependencies	The runtime must be active and listening to module execution requests.

UC3: Check Runtime Health

This use case's goal is to check if the runtime is still active and listening to requests. It is important to check the health of the instance because it involves maintaining the runtime data in the system that uses it.

Table 3.3: Check runtime health use case specification

Parameter	Value
Acceptance Criteria	The system receives response from the runtime
Input	Runtime ID
Output	Runtime health message
Dependencies	The system must know the runtime ID.

3.1.2 Non-Functional Requirements

To define the non-functional requirements of this project, the FURPS model was used. FURPS is an acronym for functionality, usability, reliability, performance and supportability.

- **Functionality**
  1. The runtime must execute the WebAssembly modules in a sandboxed environment;
  2. The runtime must be able to receive and publish messages to a MQTT message broker;
  3. The runtime must be able to support multiple WebAssembly runtime engines and provide a simple way to integrate more;
- **Usability**
  1. The runtime must be totally automatic, from the registration process to the module execution;
  2. The runtime must have a user interface to visualize the module requests that arrive to the browser and check the module output;
- **Reliability**
  1. The runtime should be prepared for any eventual error during the execution of modules and notify the runtime manager when an exception is thrown;
- **Performance**
  1. The runtime must prepare the environment for each module execution in under 1 second. This includes processing the request message and setting up the sandboxed environment;
- **Supportability**
  1. The runtime must be supported in all major browsers.

3.2 General Concepts and Design Choices

The project is essentially a client-side web application that serves as a runtime for WebAssembly modules. The application should be able to remotely execute applications and act as a server that listens to requests.

The solution is meant to work with ARENA and ARTS, but it was developed to be supported by any other system that it would fit. This is achieved by a message format standard and a set communication protocol. The solution must also be able to support multiple WebAssembly runtime engines and easily integrate more as the time goes by. This section describes the main concepts that define the solution and allow for a safe, easy to implement and robust system.

3.2.1 Deployment Structure and Architecture

To understand the architecture of the solution, it is important to look at the infrastructure needed to implement the system.

The basis of the infrastructure needed to deploy the system can be sliced into 4 different components:

- The **user device (web browser)** to use the runtime. As a client-side application, all actions will be executed by the browser, therefore nothing is executed on the server;
- The **system responsible for managing the runtime instances** and features, such as trigger the execution of modules, registering active runtimes and making health checks for all instances;
- A **MQTT message broker**, responsible for all communication between the system that manages all runtime instances and the runtime itself;
- A **web server** to host the runtime binaries, so that the browser can download when the user opens the page.

Figure 3.2 represents this same deployment example for the ARENA infrastructure implementation of the runtime. In this case, the system that manages the runtime instances is ARTS (described in section 2.3.4 of the state-of-the-art).

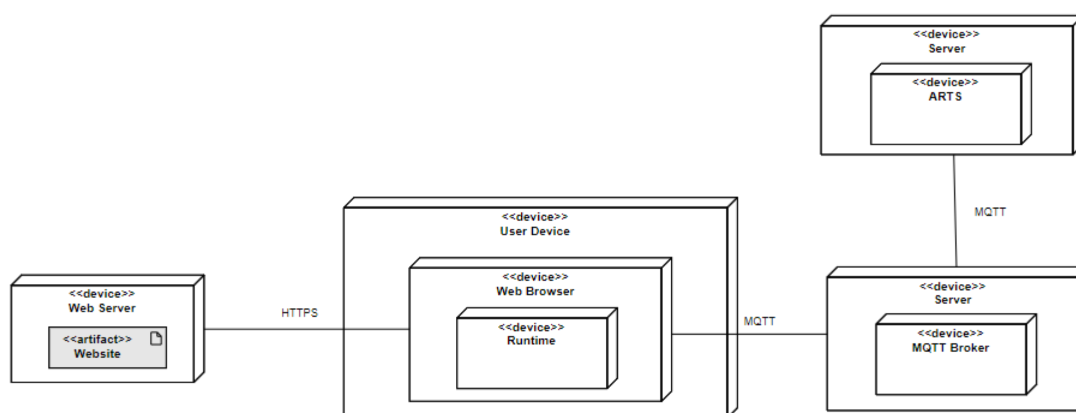

Figure 3.2: Deployment diagram example for ARENA implementation

The architecture of the runtime itself was designed to support multiple WebAssembly runtime engines like Wasmer and Runno. It was also built to allow for other engines to be easily implemented in the system as the time goes by and newer engines are developed with more features and enhancements that could be advantageous in the future. The runtime is composed of several components, each with a determined functionality:

- A user interface to visualize runtime information such as runtime configurations and modules;
- A **MQTT client** responsible for subscribing to topics and publishing messages to topics in the MQTT message broker;
- A **message manager** that is responsible for deserializing and serializing messages that come and go to the **MQTT client**;
- A **module manager** that has the responsibility of keeping the modules data and handle the instantiation of the engines.
- An **engine component** responsible for instantiating workers and handling worker messages while the modules are executing;
- A module worker that implements *Wasmer* to execute modules;

- A module worker that implements *Runno* to execute modules;
- A central **runtime component** that serves as the glue between components and handling messages.

Figure 3.3 helps visualizing the components in each runtime instance running in the user's web browser.

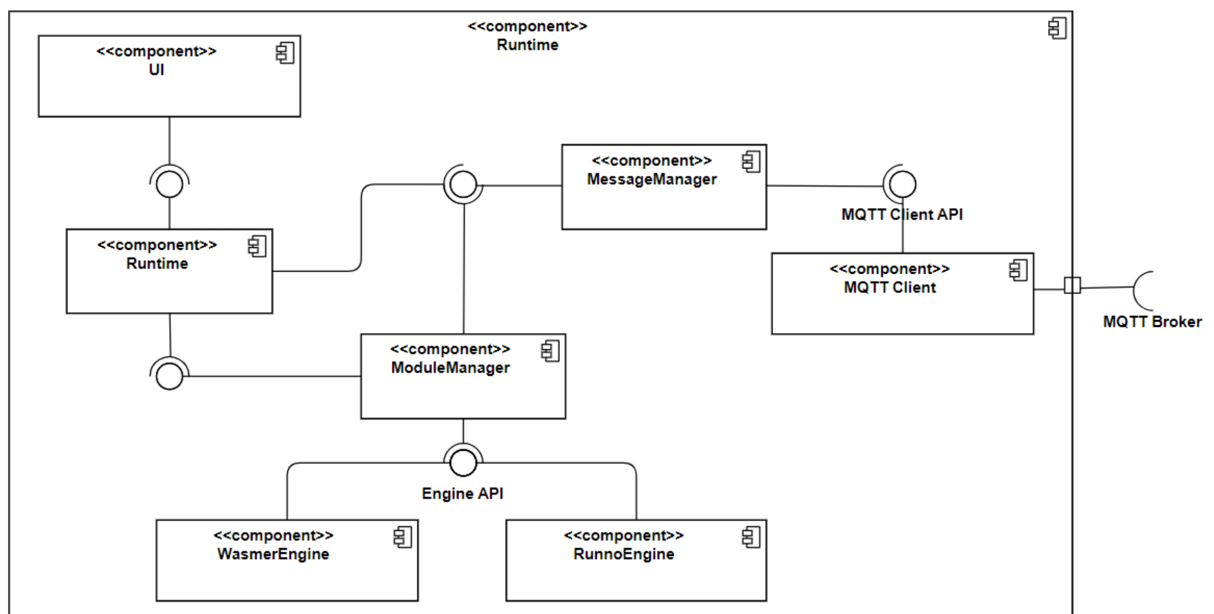

Figure 3.3: Component diagram representing the runtime architecture

3.2.2 Communication

To communicate with the application that manages the runtime (ARTS), the runtime uses MQTT as the messaging protocol. Every single instance of the runtime has a MQTT Client that connects to a message broker (also represented in figure 3.3) using a generated client identifier that is created when the instance is first loaded.

The connection is made when the user initializes the runtime and uses the configuration that is available on the web server that hosts the runtime files. Listing 3.1 shows in JSON format some of the properties that are needed to connect the runtime to the message broker.

```
{
    "mqtt_uri": "",
    "mqtt_username": "",
    "mqtt_token": "",
    "useSSL": true,
    "debug": false
}
```

Listing 3.1: Runtime Configuration Structure in JSON

- **mqtt_uri** - The URI to the MQTT message broker. As the connection is made from a web application running on a web browser, the type of connection in the URI must be via WebSocket. The format of the uri is always similar to *wss://host:port/mqtt*,

with *host* being the URL to the host of the broker, and *port* being the port where the broker is hosted;

- **mqtt_username** - The name of the user that will be used to authenticate in the message broker;
- **mqtt_token** - The password of the user that will be used to authenticate in the message broker;
- **useSSL** - A boolean property that specifies if SSL is used to connect to the broker;
- **debug** - A boolean property that specifies if detailed logs are printed to the console for debugging purposes;
- **client_id** - Although it is not configurable, the client_id is mandatory for an application to connect to a MQTT message broker. It is automatically generated in the runtime instance startup and serves as the runtime identifier as well.

A MQTT topic is a string used to filter client messages. In this system it is used as a thread to publish messages and separate each runtime instance. Every runtime works with three different MQTT topics: one common to all instances and two other specific to the instance itself.

- **runtime/register** - this topic is common to all runtime instances and is meant to be subscribed by both instances and the manager (ARTS). The messages published to this topic are related to the registration of the runtimes;
- **runtime/client_id/module** - each runtime instance has a topic of their own. All instances subscribe to this topic in order to receive orders from the manager to execute modules. What makes each topic unique is the client_id in the name, which is replaced by the runtime identifier;
- **runtime/client_id** - this topic is used for health checks to a specific runtime.

The runtime only handles messages that follow a specific standard and messages published to the topics it subscribes to. There are different types of messages for different use cases, which will be described further in this document.

3.2.3 Messages

The portability of the runtime extends other applications outside ARENA, which means it does not contain specific aspects that would break when using it in another system. To achieve such portability, a messaging standard was created so that developers that need a system like this could easily build a platform that manages the runtime resources and features. All messages payloads that the runtime handles must comply with the standard. Figure 3.4 displays a visual representation of the message format as an inheritance relationship between the main class Message and subclasses that represent different message types.

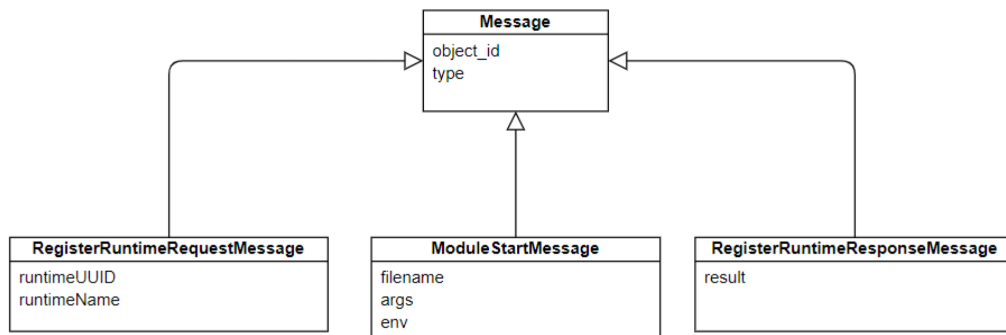

Figure 3.4: Messaging standard as an inheritance relationship

The MQTT messages have a readable payload. Although JSON was used in the implementation, the format is flexible and could be replaced by other serialization alternatives such as XML. The specific format is not critical, as what matters is the content and structure of the message payload.

Every single message that is sent or received by the runtime has an *object_id* to identify the message itself and posterior responses to it, as well as a *type* attribute that identifies the purpose of the message. The attribute *data* varies in type and properties, depending on the type of message and what it represents.

3.3 Processes

This section goes use case by use case describing and analysing the processes and designing the routines to be implemented by the runtime. From the registration of the runtime to the execution of modules, each process is thoroughly represented with UML diagrams and detailed with descriptions.

3.3.1 UC1: Register Runtime

To register a runtime, the system needs to let the manager know that it was just created. As this use case is triggered by the runtime page being opened, the main actor is the web browser user. The main idea of the process is to publish a message to the runtime/register topic with the runtime data, and then wait for a response from the manager to let the runtime know it was registered and that it can finally create and subscribe to its own module topic. Figure 3.5 and 3.6 are process diagrams that describe two parts of the use case process: the publishing of the register message and the response to such registration request.

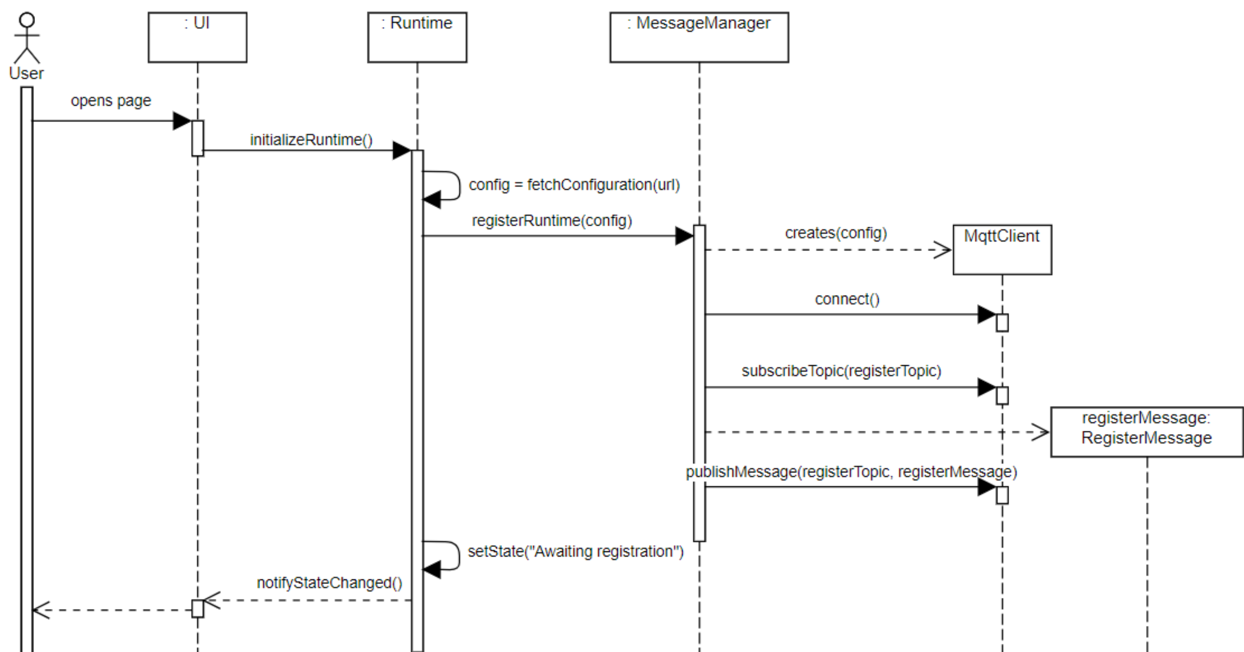

Figure 3.5: Process Diagram for UC1 (register request)

1. **The user opens the runtime page**. This is what triggers the application to start the process;
2. **The runtime fetches the MQTT configuration from the server**;
3. **The message manager creates an instance of the MQTT client**. This instance is responsible for connecting to the broker and notify the runtime when a message arrives;
4. **The message manager uses the MQTT client to connect to the message broker**;
5. **The message manager subscribes to the register topic**. This is done because the response for the register request is sent to the same topic of the request;
6. **The message manager creates the register request message with its own data** and calls the MQTT client to publish it;
7. **The state of the runtime is changed** and the UI is notified to display this information;

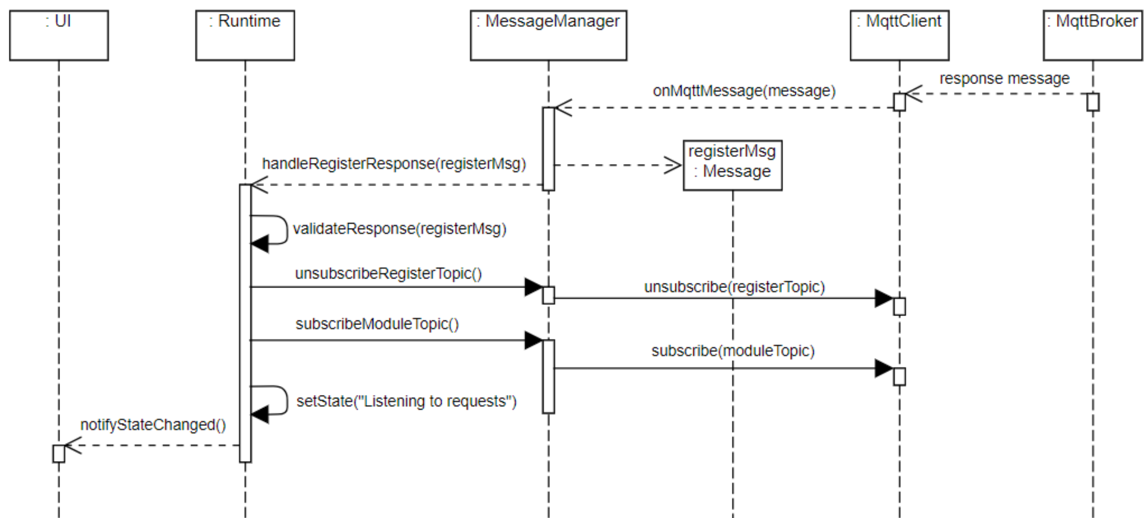

Figure 3.6: Process Diagram for UC1 (handle manager response)

1. **The response message arrives in the message broker** and the MQTT client invokes a callback to the message manager, in order to handle it;
2. **The message manager takes the payload of the message and parses it to an object** and invokes a callback for the runtime to handle validations;
3. **The runtime checks if the response status is successful** and if the message identifier is the same as the request message's. This is what guarantees the runtime that this message is a response for the initial register request;
4. **The message manager unsubscribes to the register topic**;
5. **The message manager subscribes to the runtime's module topic** in order to start listening for module execution requests from the manager;
6. **The state of the runtime is changed** and the UI is notified to display this information;

At this point, the use case ended successfully and the runtime instance is subscribed to the designated module topic in order to receive requests from the manager.

3.3.2 UC2: Start Module

This use case occurs when the manager application (ARTS, for example) publishes a message to the module topic of the runtime instance. Figure 3.7 represents the process of this use case in a diagram.

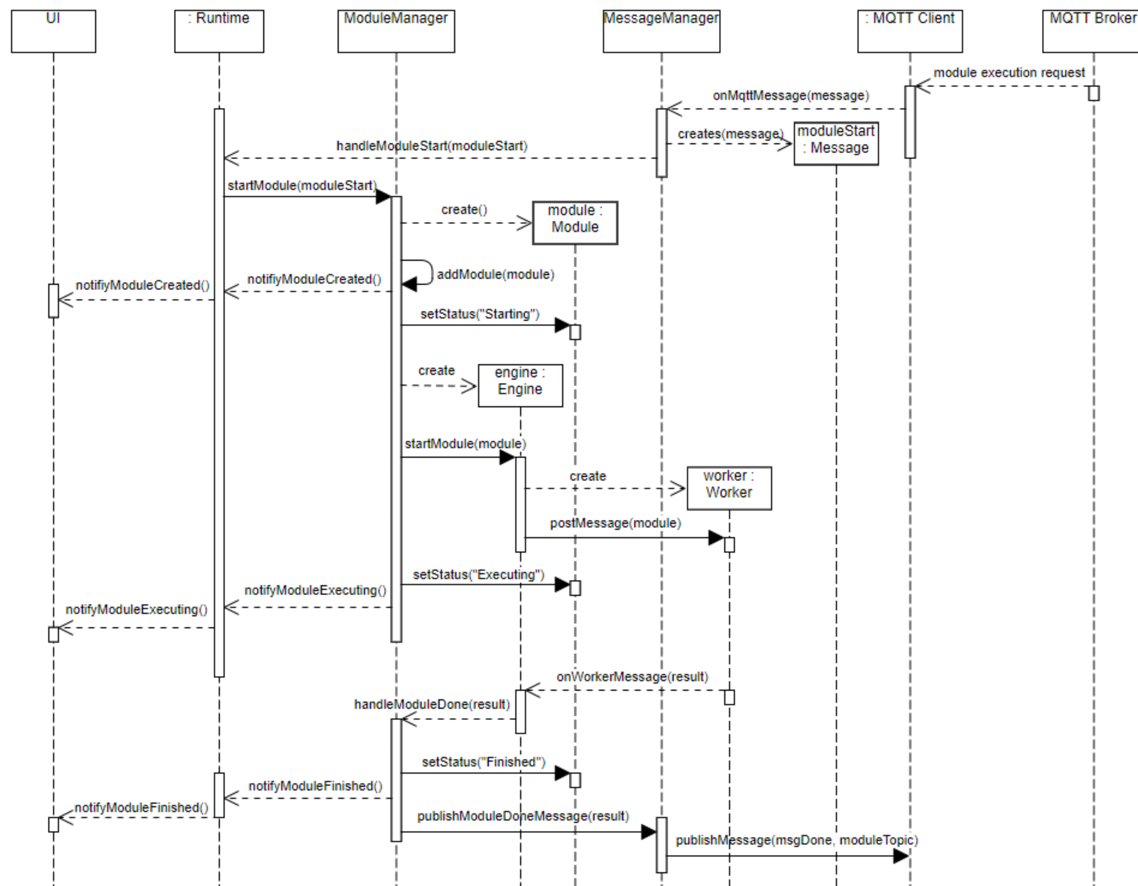

Figure 3.7: Process Diagram for UC2 (start module)

1. **A start module message is published to the module topic of the runtime**;
2. **The message manager takes the payload of the message and parses it to an object** and invokes a callback for the runtime to handle it;
3. The runtime invokes the module manager, which with the data from the message, **creates a module object and saves it in memory**. The UI is also notified that a new module was created;
4. The module manager instantiates an engine to handle worker creation and messaging.
5. As the message specifies what engine to use, **the engine creates a Worker with the specified engine**;
6. **The engine sends a message to start the worker** and consequently, execute the module;
7. The module status changes to "Executing" and the UI is notified to display the information;
8. **When the module is done executing**, the worker will post a message with the result of the module execution;
9. **The engine invokes a callback** for the module manager to handle the termination of the module by changing the status and notifying the runtime;

10. **The runtime orders the message manager to publish a message informing that the module is done executing** and notifies the UI.

Details of what is done inside the worker will be detailed further in the document, specifically in the implementation section.

3.3.3 UC3: Check Runtime Health

This use case is triggered when the manager application publishes a message to the *runtime/client_id* topic. A simple message that that requires a response back to check if the runtime is still active and listening to requests. Figure 3.8 is a diagram representing this process.

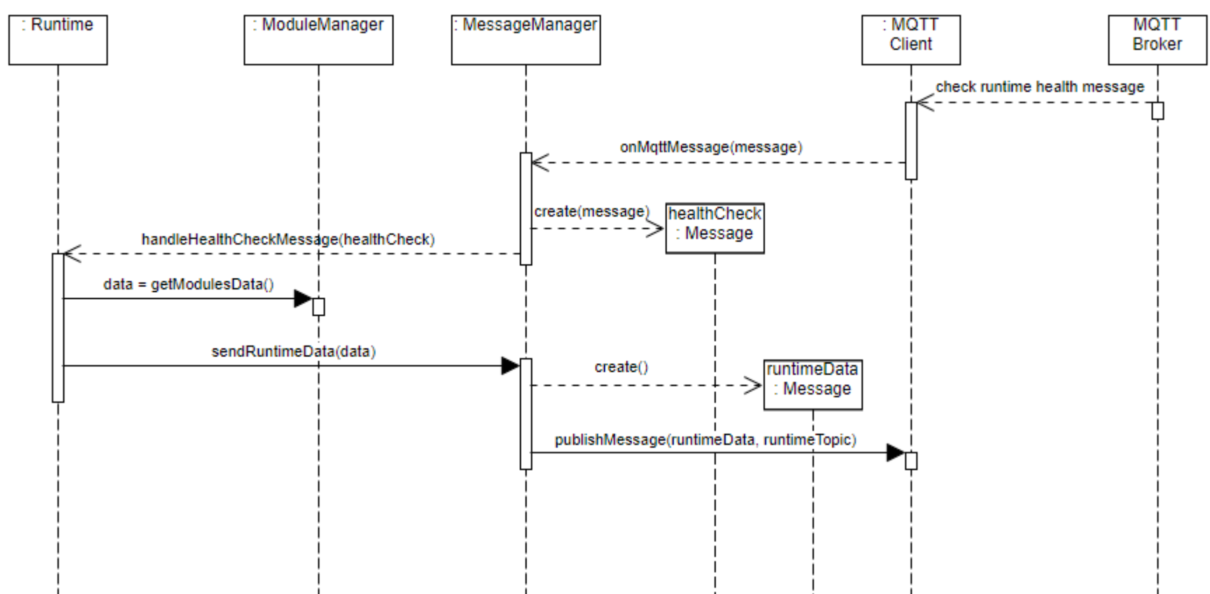

Figure 3.8: Process Diagram for UC3 (check runtime health)

1. **A health check message arrives in the MQTT Client**;
2. **The message manager takes the payload of the message, parses it to an object** and invokes a callback for the runtime to handle it;
3. **the runtime obtains the modules data** from the module manager;
4. **The message manager builds and publishes a message** with the same identifier to let the manager know that it is a direct response to the health check request.

This way, the manager is always synchronized with the health of the runtime instance. If it does not get a response back, it will assume that the runtime was closed and disposed.

3.4 Implementation

This section is dedicated to the description of implementation aspects. It will detail how to WebAssembly modules are instantiated and executed, as well as the user interface built to easily interact with the runtime data.

3.4.1 Module execution

In the core of the solution lies a system that executes WebAssembly modules, supported by two different implementations that are chosen when the manager sends the message to the runtime to start the application. The two implementations use Wasmer and Runno JavaScript libraries and both of them allow the runtime to support modules that were compiled using the WASI standard.

As defined in section 3.3.2, every module is executed by its own JavaScript worker that runs in a separated thread, which allows to run multiple WebAssembly modules at the same time. The message that triggers the module execution has a property called *engine* that represents an enumerate with 2 possible values: "wasmer" or "runno". Based on the *engine* property, the system chooses what type of worker to instantiate and because both types of workers expect the same input data, the worker startup message does not need to know what kind of implementation it is using.

The data sent to the web worker includes all necessary properties to execute a WebAssembly module:

- **filename** - the URL/path to the WebAssembly (.wasm) binary file. This is the way the runtime knows where to fetch the binaries from;
- **args** - this is an array of arguments to pass into the entrypoint function of the module;
- **env** - this is a set of keys that can be passed as environment variables to the module sandbox;
- **uuid** - the identifier of the module being executed, for further identification when the worker sends a message related to the module execution.

The component responsible for the creation of the workers and communication is the engine. Every module request has its own engine instance that not only chooses what implementation to use, but also handle messages from those workers, for example when a module is done executing, the engine will receive the message and notify the runtime whether the execution was successful or not. The module data comes from MQTT message payload, which was published to the runtime modules topic by the manager (ARTS).

After the engine is created, the runtime will give order to the engine to start the module, which means the engine will post a message to the worker with the module data. At this moment, the worker will boot up and start preparing the sandboxed environment to execute the WebAssembly application and start the module. When the worker is finished, it will post a message that will be caught by the engine, which in turn will terminate the worker and let the runtime know if the module execution was successful or not through two different callbacks.

Next is a detailed explanation of both implementations of the workers, with specific concepts and step by step instructions of what each implementation does.

Wasmer implementation

The Wasmer implementation involves the use of a WASI object that is created to configure the environment needed to support WebAssembly modules that require the use of WASI functionalities. This object is created with the environment variables that come with the module data, as well as the arguments of the application. The object also requires a set of

bindings that are available in the Wasmer SDK, to specify if the module is being ran on a web browser or in a Node application.

The chosen version of the Wasmer JavaScript library has a particular aspect related to the treatment of the WebAssembly binary after being fetched. Wasmer provides a way to lower the bytes of the binary in order to boost performance while executing the module through the *@wasmer/wasm-transformer* package. Wasmer also provides a wrapper to simulate a filesystem environment, in case the module needs to manipulate files in memory.

The instantiation of the module is done by the standard WebAssembly JavaScript library, but it is started via the created WASI object that takes the instance as a parameter. When instantiating the module, a set of options must be specified: the WASI imports generated based on the binaries that allow the module to support WASI calls and a set of JavaScript functions implemented by the runtime that the module can call during execution.

Runno implementation

The Runno's implementation also makes use of a WASI class to create an object that calculates the WASI imports that the module needs. While some of the properties are the same as in the Wasmer implementation (the environment variables and module arguments), the object also needs three callbacks to handle the standard output, error and user prompts.

Opposed to the Wasmer implementation, the instantiation of the module is done by using the *instantiateStreaming* function, which takes the binaries of the WebAssembly module and an object that contains all necessary imports including WASI features and specific runtime functions. The WebAssembly instance is then used to startup the module.

3.4.2 Runtime User Interface

Given that the runtime operates as a web application within a web browser, a user interface was developed to assist users in visualizing both runtime data and module data effectively. It was built primarily as a helper and monitoring station to check the runtime status, all the modules that were requested by remote applications, as well as exceptions thrown during execution and possible outputs.

Figure 3.9 represents the user interface that appears when someone uses the runtime in their respective browser. There is also a breakdown of the interface elements and what data it displays.

Initialize Runtime

Runtime

Runtime Status: **Listening to requests**

Runtime Info

Name:**Nuno's Runtime**

UUID:**client_30132**

Modules Topic:**runtime/client_30132/module**

Register UUID:**73fb3d5f-4c4b-46f7-8d34-7cc9f6bd15f1**

Modules

Name	Filename	Engine type	Status	Output
Fibonacci	https://nunowasmruntime.l...	wasmer	Finished	
Fibonacci (Runno)	https://nunowasmruntime.l...	runno	Finished	

Figure 3.9: Runtime user interface with example modules

The user interface can be divided in three sections: the header, the runtime info section and the modules grid.

- **Header**: the header contains a button that once clicked, triggers the registering of the runtime process. This process, also described in section 3.3.1, manipulates the status of the runtime which is displayed on the far left of the header. The status default value is "Off", but once the registration begins, it evolves into "Awaiting confirmation" and finally "Listening to requests" as shown in the figure above. In the center of the header there is a simple title;
- **Runtime information section**: this section is dedicated to the display of essential runtime information such as it's unique identification string, the name of the topic used to publish module execution requests and the UUID of the registration request;
- **Modules section**: this section it's composed by a single grid that displays every module request that arrives in the module topic of the runtime instance. The grid displays the name of the modules, the path to the binaries of the WebAssembly module, the type of implementation chosen to execute the module, the status of the module and a special column with a button that once clicked shows a pop-up with the module's output.

Chapter 4

Benchmarking

This chapter goes through the performance results of the solution and processes that it involves. It begins with a description of the methodology followed to gather the data and the way such data is analyzed. Then, several processes of the solution are benchmarked with cumulative distribution functions, like the registration of the runtime and multiple module executions.

4.1 Methodology

This section describes the methodology used to benchmark the most important processes of the runtime. The system should be able to guarantee fast performance times to deliver the best experience not only to the ARENA users (for virtual reality programs) but also other applications executed by the runtime.

Every process detailed in this chapter is benchmarked through the use of cumulative distribution functions to visualize the proportion of executions that complete at a given time and highlight the variability of the runtime performance.

The data used to analyze the runtime performance was obtained through the automatic execution of all the processes 100 times for each process and written in a CSV file. The graphics were built using Python with the *seaborn* module.

The processes related to the execution of modules and sandbox setup for modules are divided for each engine implementation: wasmer and runno. Furthermore, three different modules were used to benchmark both engines, each performing different tasks such as arithmetic operations and encryption. For every module that the runtime was benchmarked, there are two processes that were analyzed separately for a deeper understading:

- the module sandbox setup times, where the environment is prepared to execute the module, including the fetching of the WebAssembly binaries, arguments arrangements and WASI bindings;
- the execution of the modules itself, starting from the moment the engines call the module entrypoint and finishes executing the functions.

4.2 Runtime Registration Benchmarking

The benchmarking of the runtime registration process was made by placing a timer before the runtime starts to build the first registration message and after it sends such message

connected MQTT broker. This can be influenced by the connection to the broker, depending where such is located and internet connection.

From the one hundred times the process was executed, the obtained values go from approximately 220 milliseconds up to 360 milliseconds. The cumulative distribution function represented in figure 4.1 shows that this process is most likely to execute in under 320 milliseconds, but it is unlikely that this time is faster than 240 milliseconds. The steep line between 260 milliseconds and 300 milliseconds means that a significant portion of the execution times was concentrated around the median value.

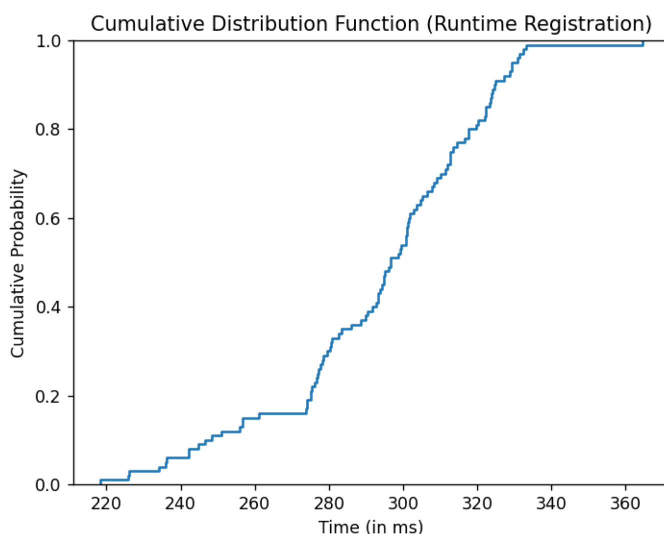

Figure 4.1: Cumulative distribution function for runtime registration benchmarks

For this process, the average execution time is 293 milliseconds and the median is 297 milliseconds.

4.3 Module Request Message Processing Benchmarking

In this case, the timer starts as soon as the MQTT message arrives at the message manager for processing and stops when the worker is notified to start it's work executing the module. It covers payload serialization, processing and the creation of the worker that executes the module.

The cumulative distribution function represented in figure 4.2 shows that there were times registered up to 18 milliseconds, but there is a much higher probability that this process executes faster than 5 milliseconds. The steep portion between 1 millisecond and 5 milliseconds reveals that most of the runs were executed between these values, and the flat portion located higher after 6 milliseconds reveal some outliers.

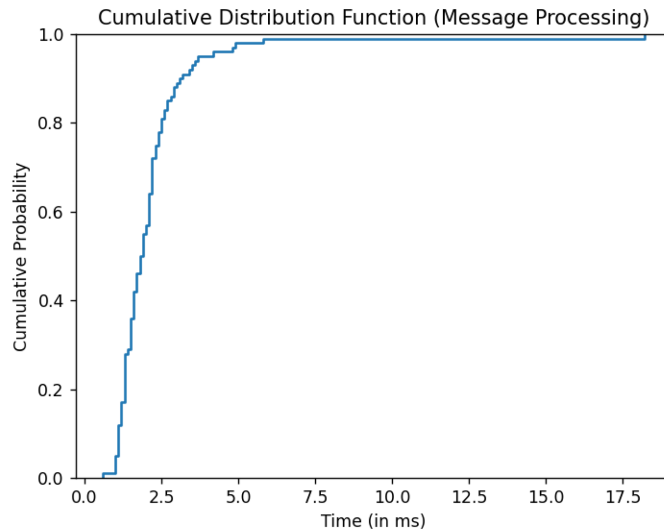

Figure 4.2: Cumulative distribution function for message processing benchmarks

For this process, the average execution time is 2.2 milliseconds and the median value is also 1.9 milliseconds.

4.4 Module Execution Benchmarking - Box

The box module is a series of tests that combine bitwise and arithmetic operations [50].

4.4.1 Module Setup

Setting up the environment for this module took on average 21.9 milliseconds for Wasmer and 11.6 milliseconds for Runno. The cumulative distribution functions for both engines is represented in figure 4.3 and it shows that Runno is not only more consistent, but also faster considering that the probability of this process executing in less than 18 milliseconds is very high compared to the approximately 25% chance of the Wasmer executing this process in the same time.

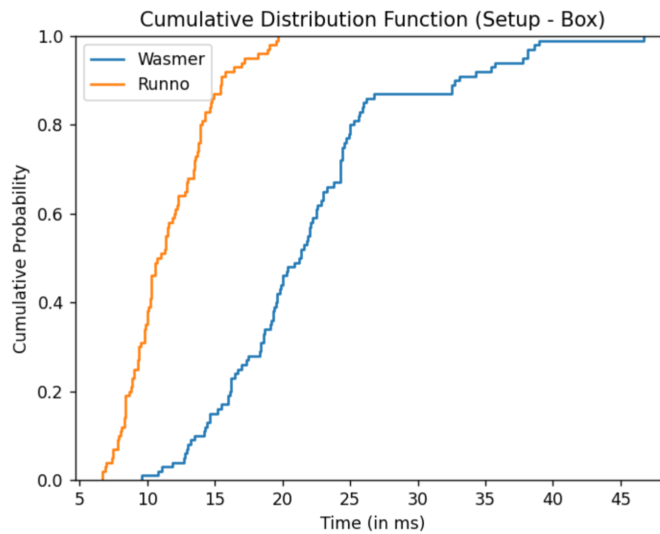

Figure 4.3: Cumulative distribution functions for the box module setup times

The median value in this process was 21.3 milliseconds for Wasmer and 10,8 milliseconds for Runno.

4.4.2 Module Execution

Contrary to the setup of the module, executing it has revealed to be closer. Runno is more consistent with values between 475 milliseconds and 650 milliseconds opposed to the wider range of Wasmer execution times, but the counter part managed to score times faster than 475 milliseconds. The cumulative distribution functions of figure 4.4 reveal that by looking where the curves cross both engines have the same probability of scoring times faster than 575 milliseconds and other instances.

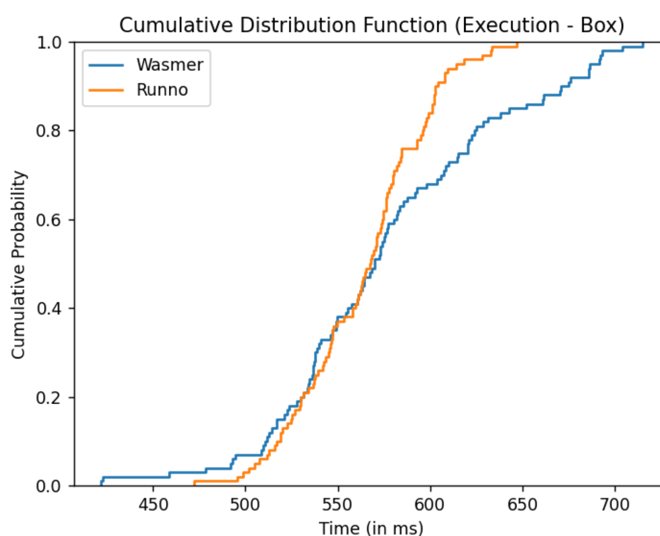

Figure 4.4: Cumulative distribution functions for the box module execution times

The average execution time for the box module is 576.7 milliseconds for Wasmer and 563.6 for Runno. As for the median value, Runno got 567.5 milliseconds and Wasmer got 569.8 milliseconds.

4.5 Module Execution Benchmarking - Authentication

The module used to benchmark the runtime in this section is based on software implementation of the SHA-2 hash function [50].

4.5.1 Module Setup

For the setup of the environment for the authentication module, the cumulative distribution function of figure 4.5 reveals that both curves are quite similar in shape. This is due to the fact that they both share similar probabilities of executing the process in less or equal time. Both engines are very consistent in times between 10 milliseconds and 45 milliseconds, with some outliers with low chances of slower times revealed by the flat portion between 50 milliseconds and the maximum scored time of 270 milliseconds.

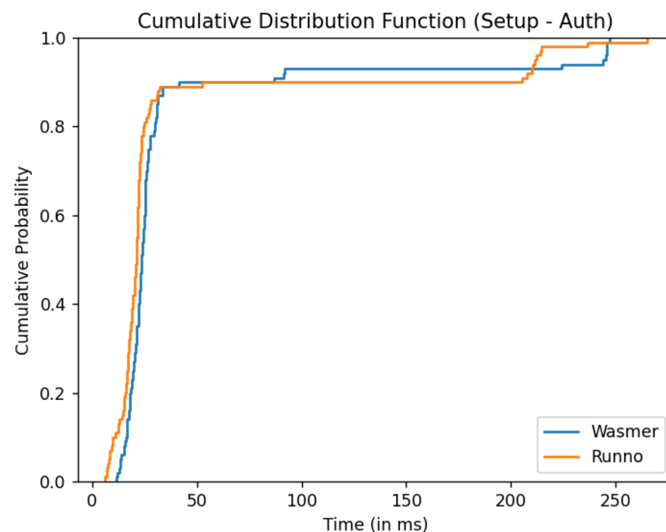

Figure 4.5: Cumulative distribution functions for the authentication module setup times

The average setup time for the authentication module is 40.2 milliseconds for Wasmer and 39.4 for Runno. As for the median value, Runno got 23.7 milliseconds and Wasmer got 21.1 milliseconds.

4.5.2 Module Execution

Similar to the environment setup for the authentication module, the cumulative distribution functions visualized in figure 4.6 reveal two close curves. Both lines have a similar range of values and have a steep between the execution times of 60 milliseconds and 75 milliseconds. The two curves separate in the 50 to 60 milliseconds range, where it is understood that Wasmer has a higher chance of executing the module in less than the values in this range compared to Runno.

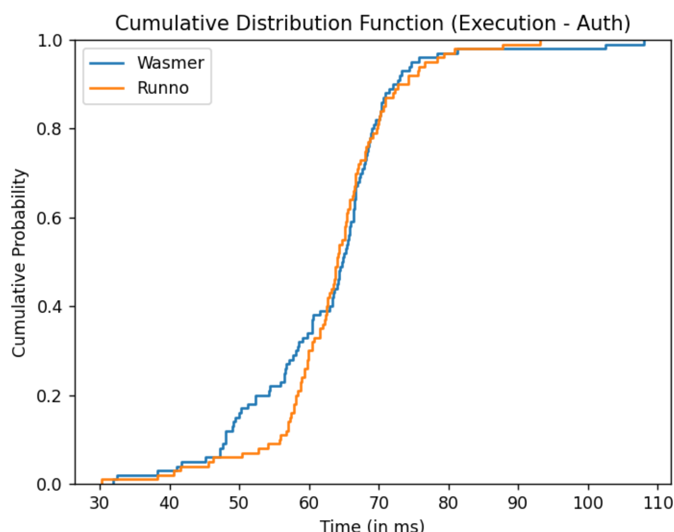

Figure 4.6: Cumulative distribution functions for the authentication module execution times

The average execution time for the authentication module is 62.5 milliseconds for Wasmer and 63.7 for Runno. As for the median value, Runno got 64.1 milliseconds and Wasmer got 64.8 milliseconds.

4.6 Module Execution Benchmarking - AEAD

The authenticated encryption tests are based on a small function that is called multiple times with different parameters [50].

4.6.1 Module Setup

The cumulative distributed functions of figure 4.7 for the environment setup of the AEAD module reveal that Runno was very consistent throughout the hundred executions of the process (with values between 7 and 25 milliseconds) and some major outliers above 400 milliseconds for Wasmer, which could be caused by lack of stability in the internet connection at the time of data gathering, mainly in the fetching of the WebAssembly binaries. The probability of scoring faster times than those above 400 milliseconds when using Wasmer are higher than 90%, but it doesn't change the fact that Runno had a more consistent set of runs.

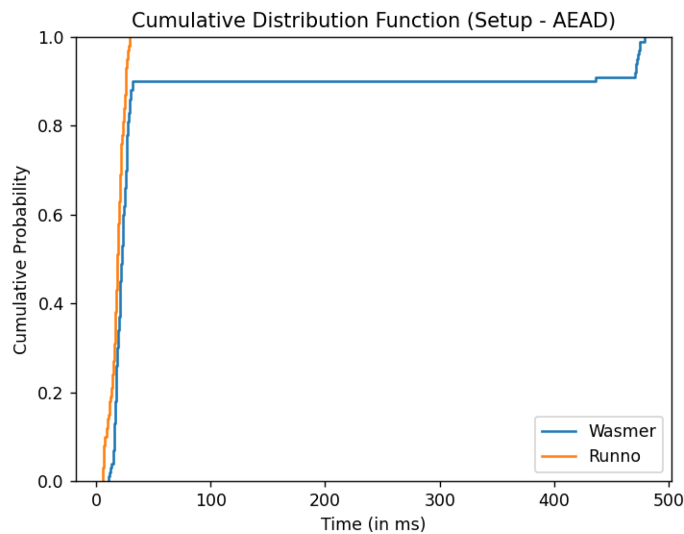

Figure 4.7: Cumulative distribution functions for the AEAD module setup times

The average setup time for the AEAD module is 66.6 milliseconds for Wasmer and 18.3 for Runno. As for the median value, Runno got 18.8 milliseconds and Wasmer got 22.4 milliseconds.

4.6.2 Module Execution

The execution of the authenticated encryption tests revealed that both engines had wide ranges of times, with Wasmer scoring the faster and slowest time. The cumulative distribution functions, represented in figure 4.8, show that both engines have the same probability of scoring a time faster or equal to approximately 210 milliseconds and also that Wasmer has a higher chance than Runno of executing the module in times in the range of 160 milliseconds to 200 milliseconds.

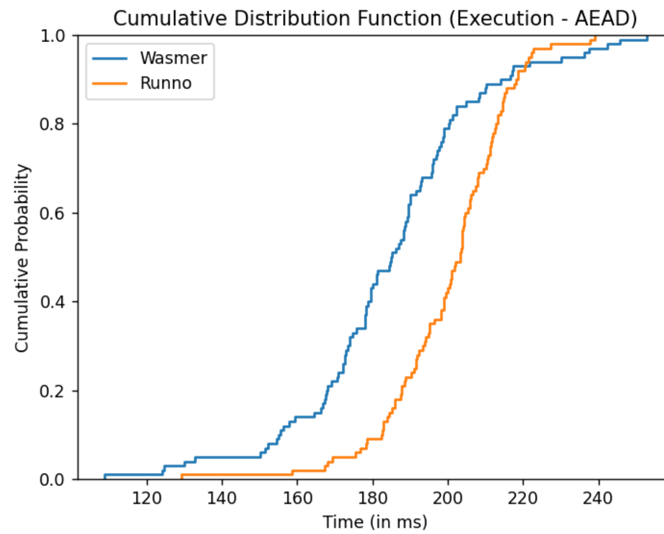

Figure 4.8: Cumulative distribution functions for the AEAD module execution times

The average execution time for the AEAD module is 184.7 milliseconds for Wasmer and 200.4 for Runno. As for the median value, Runno got 203.4 milliseconds and Wasmer got 185 milliseconds.

Chapter 5

Conclusion

The need for a computation offload solution motivated the work done in this project. Bringing the execution of applications closer to the users is a way to reduce latency and boost performance for systems like the ARENA. This project resulted in a browser-based WebAssembly runtime that allows applications to be distributed in a network of different devices, and following a concise methodology the goals that were setup for this project were successfully fulfilled.

An analysis of multiple WebAssembly runtimes with comparisons between them led to the choice of using two different engines and the decision to design the solution to support various runtimes with an architecture that would allow the implementation of more in the future without difficulty.

The browser-based runtime was successfully designed and implemented, together with a dashboard to interact with it's features such as the registration of the runtime, gather data from the modules and visualization of module executions status. The solution communicates through the use of the MQTT protocol and a standard for structuring the content of the messages payloads was also designed to provide stability and reliability between components. The runtime proved to work in high performance, thanks to a benchmarking process that analyzed the main use cases and processes, including the runtime registration and the execution of three different WebAssembly modules, which most revealed to be executed in an average under 500 milliseconds.

The undertaken research and development efforts also allowed for a reliable and expandable project, that could be continued in future works like the implementation of the runtime in the ARENA frameworks to dispatch applications to the edge, the addition of more execution engines for WebAssembly modules and also the exploration of environments outside the browser. In conclusion, following the established process and methodology, all the goals set for this project were successfully achieved and the project provides a solid foundation for future work.

Bibliography

[1] P Schlummer et al. "Seeing the unseen—enhancing and evaluating undergraduate polarization experiments with interactive Mixed-Reality technology". In: *European Journal of Physics* 44.6 (Sept. 2023), p. 065701. doi: 10.1088/1361-6404/acf0a7. url: https://dx.doi.org/10.1088/1361-6404/acf0a7.

[2] Mario Lorenz, Sebastian Knopp, and Philipp Klimant. "Industrial Augmented Reality: Requirements for an Augmented Reality Maintenance Worker Support System". In: *2018 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*. 2018, pp. 151–153. doi: 10.1109/ISMAR-Adjunct.2018.00055.

[3] Christoph Bichlmeier et al. "Stepping into the operating theater: ARAV — Augmented Reality Aided Vertebroplasty". In: *2008 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*. 2008, pp. 165–166. doi: 10.1109/ISMAR.2008.4637348.

[4] Z. Jonny Kong et al. "AccuMO: Accuracy-Centric Multitask Offloading in Edge-Assisted Mobile Augmented Reality". In: *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. ACM MobiCom '23. Madrid, Spain: Association for Computing Machinery, 2023. isbn: 9781450399906. doi: 10.1145/3570361.3592531. url: https://doi.org/10.1145/3570361.3592531.

[5] Andrea Di Domenico et al. "A Network Analysis on Cloud Gaming: Stadia, GeForce Now and PSNow". In: *Network* 1.3 (2021), pp. 247–260. issn: 2673-8732. doi: 10.3390/network1030015. url: https://www.mdpi.com/2673-8732/1/3/15.

[6] Dar, Daks Ravindran, and Shahidul Islam. "Fog-based Spider Web Algorithm to Overcome Latency in Cloud Computing". In: *Iraqi Journal of Science* 61 (July 2020), pp. 1781–1790. doi: 10.24996/ijs.2020.61.7.27.

[7] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. "Decentralized Resource Auctioning for Latency-Sensitive Edge Computing". In: *2019 IEEE International Conference on Edge Computing (EDGE)*. 2019, pp. 72–76. doi: 10.1109/EDGE.2019.00027.

[8] Nuno Pereira et al. "ARENA: The Augmented Reality Edge Networking Architecture". In: *2021 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. 2021, pp. 479–488. doi: 10.1109/ISMAR52148.2021.00065.

[9] Ilija Hadžić, Yoshihisa Abe, and Hans C. Woithe. "Edge Computing in the EPC: A Reality Check". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. SEC '17. San Jose, California: Association for Computing Machinery, 2017. isbn: 9781450350877. doi: 10.1145/3132211.3134449. url: https://doi.org/10.1145/3132211.3134449.

[10] Julien Gedeon et al. "What the Fog? Edge Computing Revisited: Promises, Applications and Future Challenges". In: *IEEE Access* 7 (2019), pp. 152847–152878. doi: 10.1109/ACCESS.2019.2948399.

[11] Mahadev Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (2017), pp. 30–39. doi: 10.1109/MC.2017.9.

[12] Xuejun Li et al. "COMEC: Computation Offloading for Video-Based Heart Rate Detection APP in Mobile Edge Computing". In: 2018, pp. 1038–1039. doi: 10.1109/BDCloud.2018.00152.

[13] Liangcai Fang et al. "A Mobile Edge Computing Architecture for Safety in Mining Industry". In: 2019, pp. 1494–1498. doi: 10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00269.

[14] Giuseppe Avino et al. "A MEC-based Extended Virtual Sensing for Automotive Services". In: *2019 AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE)*. 2019, pp. 1–6. doi: 10.23919/EETA.2019.8804512.

[15] Tabasum Mirza, Neha Tuli, and Archana Mantri. "Virtual Reality, Augmented Reality, and Mixed Reality Applications: Present Scenario". In: *2022 2nd International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE)*. 2022, pp. 1405–1412. doi: 10.1109/ICACITE53722.2022.9823482.

[16] Paul Milgram and Fumio Kishino. "A Taxonomy of Mixed Reality Visual Displays". In: *IEICE Trans. Information Systems* vol. E77-D, no. 12 (Dec. 1994), pp. 1321–1329.

[17] Nicoletta Sala. "Virtual reality, augmented reality, and mixed reality in education: A brief overview". In: *Current and prospective applications of virtual reality in higher education* (2021), pp. 48–73.

[18] *HoloAnatomy*. Jan. 2023. url: https://case.edu/holoanatomy/?culture=en-us&country=us.

[19] *zSpace Website*. url: https://zspace.com/solutions.

[20] Gaurang Bansal et al. "Healthcare in Metaverse: A Survey on Current Metaverse Applications in Healthcare". In: *IEEE Access* 10 (2022), pp. 119914–119946. doi: 10.1109/ACCESS.2022.3219845.

[21] Gabriel Evans et al. "Evaluating the Microsoft HoloLens through an augmented reality assembly application". In: *Degraded Environments: Sensing, Processing, and Display 2017*. Ed. by John (Jack) N. Sanders-Reed and Jarvis (Trey) J. Arthur III. Vol. 10197. International Society for Optics and Photonics. SPIE, 2017, p. 101970V. doi: 10.1117/12.2262626. url: https://doi.org/10.1117/12.2262626.

[22] Niantic. *Pokémon GO*. [Accessed 20-08-2024]. url: https://pokemongolive.com/.

[23] *Unity Website*. Dec. 2023. url: https://unity.com/.

[24] *Apple's ARKit Documentation Website*. Dec. 2023. url: https://developer.apple.com/documentation/arkit.

[25] *Google's ARCore Website*. Dec. 2023. url: https://developers.google.com/ar.

[26] Polar-Kev. *MRTK2-Unity Developer Documentation - MRTK 2*. url: https://learn.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/mrtk2/?view=mrtkunity-2022-05.

[27] *AR.js Documentation*. url: https://ar-js-org.github.io/AR.js-Docs/.

[28] Diego Marcos. *A-Frame: Web framework for building virtual reality experiences.* url: https://github.com/aframevr/aframe/.

[29] Three.js Developers. *Three.js Library*. Online. Accessed: 2020-10-20. Apr. 2010. url: https://threejs.org/.

[30] *Augmented Reality Edge Network Architecture*. Dec. 2023. url: https://arenaxr.org/.

[31] Carl Hewitt, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[32] João De Macedo et al. "On the Runtime and Energy Performance of WebAssembly: Is WebAssembly superior to JavaScript yet?" In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 2021, pp. 255–262. doi: 10.1109/ASEW52652.2021.00056.

[33] Andreas Haas et al. "Bringing the Web up to Speed with WebAssembly". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 185–200. issn: 0362-1340. doi: 10.1145/3140587.3062363. url: https://doi.org/10.1145/3140587.3062363.

[34] Apr. 2022. url: https://www.w3.org/TR/wasm-core-2/.

[35] Bennet Yee et al. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *IEEE Symposium on Security and Privacy (Oakland'09)*. IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009. url: http://nativeclient.googlecode.com/svn/data/docs_tarball/nacl/googleclient/native_client/documentation/nacl_paper.pdf.

[36] Alon Zakai. "Emscripten: An LLVM-to-JavaScript Compiler". In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 301–312. isbn: 9781450309424. doi: 10.1145/2048147.2048224. url: https://doi.org/10.1145/2048147.2048224.

[37] url: http://asmjs.org/.

[38] Elliott Wen and Gerald Weber. "Wasmachine: Bring IoT up to Speed with A WebAssembly OS". In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2020, pp. 1–4. doi: 10.1109/PerComWorkshops48775.2020.9156135.

[39] Jämes Ménétrey et al. "A Comprehensive Trusted Runtime for WebAssembly with Intel SGX". In: *IEEE Transactions on Dependable and Secure Computing* (2023), pp. 1–18. doi: 10.1109/TDSC.2023.3334516.

[40] Yutian Yan et al. "Understanding the Performance of Webassembly Applications". In: *Proceedings of the 21st ACM Internet Measurement Conference*. IMC '21. Virtual Event: Association for Computing Machinery, 2021, pp. 533–549. isbn: 9781450391290. doi: 10.1145/3487552.3487827. url: https://doi.org/10.1145/3487552.3487827.

[41] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. "Potential of WebAssembly for Embedded Systems". In: *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. 2022, pp. 1–4. doi: 10.1109/MECO55406.2022.9797106.

[42] Mozzilla Lin Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web*. Online. Accessed: 2024-01-20. Mar. 2019. url: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

[43] *The Wasmer JavaScript SDK*. Online. Accessed: 2024-06-25. url: https://docs.wasmer.io/.

[44] Wasmer. *WASIX - Docs*. Online. Accessed: 2024-06-25. url: https://wasix.org/docs.

[45] Wasmer. *Wasmerio/Wasmer-Js: Monorepo for Javascript WebAssembly packages by Wasmer*. Online. Accessed: 2024-06-25. url: https://github.com/wasmerio/wasmer-js.

[46] Ben Taylor. *Runno Documentation*. Online. Accessed: 2024-06-25. url: https://runno.dev/docs.

[47] *Wasmtime Documentation*. Online. Accessed: 2024-06-25. url: https://wasmtime.dev/.

[48] Intel. *WebAssembly Micro Runtime*. Online. Accessed: 2020-10-20. Apr. 2019. url: https://github.com/bytecodealliance/wasm-micro-runtime.

[49] WebAssembly CG. *The WebAssembly System Interface*. Online. Accessed: 2024-06-25. Mar. 2019. url: https://wasi.dev/.

[50] Frank Dennis. *Performance of WebAssembly runtimes in 2023*. Accessed: 2024-08-12. 2023. url: https://00f.net/2023/01/04/webassembly-benchmark-2023/.