

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Adopting Continuous Deployment of WebAssembly Applications in Internet-of-Things Systems

Joel Alexandre Vieira Fernandes



Mestrado em Engenharia Informática e Computação

Supervisor: João Pedro Dias

Co-Supervisor: André Restivo

July 17, 2024

Adopting Continuous Deployment of WebAssembly Applications in Internet-of-Things Systems

Joel Alexandre Vieira Fernandes

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Tiago Boldt Pereira de Sousa

External Examiner: Prof. Ângelo Manuel Rego e Silva Martins

Supervisor: Prof. João Pedro Matos Teixeira Dias

July 17, 2024

Abstract

In this day and age, despite not being anything new or revolutionary, the Internet-of-Things (IoT) is becoming a topic of greater interest in the world of technology. This growing enthusiasm is well-founded, as IoT plays a pivotal role in various applications we regularly engage in every day, such as smart cities and smart grid. These applications rely on the constant and updated collaboration between an increasing number of lightweight devices, the computational resources and reliability of which are particularly scarce.

In later years, Kubernetes became one of the most popular generic continuous delivery (CD) solutions for orchestrating and managing with software large-scale applications on multiple heterogeneous devices with the underlying containerization Docker technology. These kinds of CD solutions mainly focus on automating the deployment process to ensure a straightforward and efficient delivery of software changes to multiple devices. However, Docker, in particular, requires devices whose power profiles are considerably higher than desirable, as computational resources in microcontrollers used in IoT are relatively short.

More recently, though, newer solutions like Toit, Goliath, and ESPHome have emerged that help create multitasking jobs on microcontrollers, albeit without containers, and also contain CD features. Still, none of these solutions brings orchestration, scalability, and fault-tolerance to the table as heavily as we would have liked for an IoT system, keeping their approach to CD on a surface level. Some of these solutions also lack their versatility in the internal technologies and devices used, meaning there is room for improvement in this field.

Thus, this dissertation focuses on extending the capabilities of Wasmico, a platform designed for containerizing WebAssembly tasks, which have the advantage of being generated from basically any programming language, on low-end microcontrollers. Therefore, based on the Wasmico containerization architecture, the proposed solution leverages principles similar to Kubernetes to achieve its orchestration objectives. This means this work will focus on developing an orchestration system able to allocate tasks across a series of heterogeneous devices while taking into account task and device scalability and fault tolerance mechanisms, which readjust the system as it changes states, device, and task failures to conform to the standards defined by the developer.

More specifically, while exploring previous works on the field of orchestration tasks across low-end microcontrollers, a solution based on the Consensus-Based Bundle Algorithm emerged as the most promising and exciting for our particular application and devices. Therefore, with some modifications according to their found limitations to their work to fit Wasmico, we used a similar algorithm and approach to perform the allocations for our orchestration algorithm, creating the basis we needed to develop it further.

Finally, we validate and evaluate our solution regarding its ability to balance and distribute tasks across devices while regarding their physical limitations and failures. In order to perform some of these tests, we had to resort to a newly developed clone version of Wasmico based on Python, which can be run on a device with a Python interpreter. With all these tests, we were able to conclude our solution was accurate in orchestrating and distributing tasks across a variety

of devices, outperforming previous works not only in execution times but also efficiency, fault-tolerance and scalability.

Keywords: Internet-of-Things, WebAssembly, Continuous Deployment, DevOps, Orchestration, Microcontrollers

Resumo

Nos dias de hoje, apesar de não ser nada novo ou revolucionário, a *Internet-of-Things* (IoT) está a tornar-se um tema de maior interesse no mundo da tecnologia. Este crescente entusiasmo é bem fundamentado, uma vez que o IoT desempenha um papel fundamental em várias aplicações que utilizamos regularmente no nosso dia a dia, como cidades inteligentes e redes elétricas inteligentes. Estas aplicações dependem da colaboração constante e atualizada entre um número crescente de dispositivos leves, cujos recursos computacionais e estabilidade são particularmente baixos.

Nos últimos anos, o Kubernetes tornou-se uma das soluções genéricas de *continuous delivery* (CD) mais populares para orquestrar e gerir aplicações em larga escala em múltiplos dispositivos heterogêneos com a tecnologia de containerização subjacente Docker. Este tipo de soluções de CD focam-se principalmente na automação do processo de implementação no terreno para garantir uma entrega simples e eficiente de alterações do software a vários dispositivos. No entanto, o Docker, em particular, requer dispositivos cujos perfis de capacidade computacional são consideravelmente mais elevados do que o desejável, uma vez que os recursos computacionais em microcontroladores utilizados em IoT são relativamente limitados.

Mais recentemente, porém, surgiram novas soluções como Toit, Golioth e ESPHome que ajudam a criar programas multitarefa em microcontroladores, embora sem containers, e também contêm funcionalidades de CD. Ainda assim, nenhuma destas soluções traz orquestração, escalabilidade e tolerância a falhas tão fortemente como gostaríamos para um sistema em IoT, mantendo a sua abordagem a CD a um nível superficial. Algumas destas soluções também carecem de versatilidade nas tecnologias internas e dispositivos utilizados, o que significa que há espaço para melhorias neste campo.

Assim, esta dissertação foca-se em estender as capacidades do Wasmico, uma plataforma projetada para containerizar tarefas WebAssembly, que têm a vantagem de poderem ser geradas a partir de praticamente qualquer linguagem de programação, em microcontroladores de baixo custo. Portanto, com base na arquitetura de containerização do Wasmico, a solução proposta aproveita princípios semelhantes aos do Kubernetes para alcançar os seus objetivos de orquestração. Isto significa que este trabalho se concentrará no desenvolvimento de um sistema de orquestração capaz de alocar tarefas numa série de dispositivos heterogêneos, levando em consideração mecanismos de escalabilidade e tolerância a falhas de tarefas e dispositivos, que reajustam o sistema à medida que ele muda de estado, falhas de dispositivos e tarefas para conformar-se aos padrões definidos pelo desenvolvedor.

Mais especificamente, ao explorar trabalhos anteriores no campo da orquestração de tarefas em microcontroladores de baixo custo, uma solução baseada no *Consensus-Based Bundle Algorithm* emergiu como a mais promissora e interessante para a nossa aplicação e dispositivos em particular. Portanto, com algumas modificações de acordo com as limitações encontradas nos trabalhos anteriores e no Wasmico, utilizámos um algoritmo e abordagem semelhantes para realizar as alocações para o nosso algoritmo de orquestração, criando a base necessária para desenvolvê-lo ainda mais.

Finalmente, validamos e avaliamos a nossa solução quanto à sua capacidade de equilibrar e distribuir tarefas entre dispositivos, considerando as suas limitações físicas e falhas. Para realizar alguns destes testes, tivemos de recorrer a uma nova versão recentemente desenvolvida do Was-mico baseada em Python, que pode ser executada num dispositivo com um interpretador Python. Com todos estes testes, fomos capazes de concluir que a nossa solução era precisa na orquestração e distribuição de tarefas em vários dispositivos, superando trabalhos anteriores não só em tempos de execução, mas também em eficiência, recuperação após falhas e escalabilidade.

Keywords: Internet-of-Things, WebAssembly, Continuous Deployment, DevOps, Orchestration, Microcontrollers

Acknowledgements

Today, this document marks the end of my journey as a fellow student. After 17 years of education, this is the culmination of everything I have worked for, my Master's. There are numerous people who made this possible, but the ones listed below were the primary ones who made all of this happen, in no particular order.

First, I would like to thank my Mom, Irene, who, despite no longer being between us, is and always has been my pillar of support. Throughout my life, she was always the first one to help me in my studies and back my school and life decisions. Through thick and thin, even in her not-so-good days, she was always there by my side, supporting me. Without her, there is no doubt in my mind that I would not be the man I am today, and I would have never achieved this incredible milestone. This is for you, Mom. I love you. I miss you.

I want to thank my Dad, Carlos, as well, who has always been supportive of my life decisions and always keeps me on my feet. Nothing would have been possible without his support too, as he has always provided me with the best to ensure I had everything I needed to succeed in school and life.

My brother, André, is also an important piece here. His energy and silliness always help me keep a smile on my face while also keeping me feeling young and ready to face anything that poses me.

I would also like to thank my girlfriend, Raya, who, despite being all the way across the globe, has always given me the unwavering support I needed to push further in life and in my studies. Since day one, all her love and encouragement have pushed me beyond boundaries I never thought I would overcome to see the better parts of life. Thank you for always being there and putting up with me, even in my worst moments.

To my friends and family, thank you for always supporting me in my studies and helping me succeed. Undoubtedly, nothing like this would have happened without every single one of you. To the ones who doubted me, wished me the worst, and challenged me, thank you for giving me the motivation to grow and be better. Indeed, a life with no enemies, insecurities, or challenges is one with no objectives or future.

To both my supervisors, João Pedro Dias and André Restivo, thank you for your support throughout this dissertation. You have always been an inspiration to me and you were among the best teachers I ever had during these 5 years.

Joel Fernandes

*“I will struggle, I will fail,
but to surrender is a privilege I do not have.”*

Lee Sin (李青)

Contents

1	Introduction	1
1.1	Context	1
1.2	Problem	3
1.3	Motivation	3
1.4	Goals	4
1.5	Document Structure	4
2	Background	6
2.1	Internet-of-Things	6
2.2	Containerization and Orchestration	8
2.2.1	Containerization	8
2.2.2	Orchestration	10
2.3	Wasmico	11
2.4	Summary	12
3	State of the Art	14
3.1	Systematic Literature Review	14
3.1.1	Methodology	14
3.1.2	Research Questions	15
3.1.3	Search Queries	15
3.1.4	Online Databases	15
3.1.5	Results Filtering	16
3.1.6	Main Results	17
3.1.7	Extended Search	21
3.2	Grey Literature	22
3.2.1	Results	23
3.3	Results Analysis	25
3.3.1	Parameters	25
3.3.2	Systematic Literature Review	26
3.3.3	Grey Literature	26
3.3.4	Overview	27
3.4	Conclusion	28
4	Problem Statement	30
4.1	Problem	30
4.2	Desiderata	32
4.3	Hypothesis	33
4.4	Research Questions	34

4.5	Scope	34
4.6	Proposed Solution	35
4.7	Validation and Evaluation	35
4.8	Summary	36
5	Solution Overview	37
5.1	Wasmico	37
5.2	Orchestration System	40
5.2.1	Orchestration Prerequisites	40
5.2.2	Orchestration Declarative Configuration File Specification	41
5.2.3	Basic Algorithm	43
5.2.4	Consensus-Based Bundle Algorithm (CBBA)	45
5.3	UI Adaptation	50
5.4	Summary	52
6	Evaluation and Validation	53
6.1	Wasmico Python Wrapper	53
6.1.1	Overview	53
6.1.2	Limitations	55
6.2	Experiments	55
6.2.1	Device Setup	55
6.2.2	Measurements	56
6.2.3	Scenarios	56
6.2.4	Scenario 1 (S.01)	56
6.2.5	Scenario 2 (S.02)	57
6.2.6	Scenario 3 (S.03)	59
6.2.7	Scenario 4 (S.04)	60
6.2.8	Scenario 5 (S.05)	60
6.2.9	Scenario 6 (S.06)	61
6.2.10	Scenario 7 (S.07)	62
6.2.11	Scenario 8 (S.08)	63
6.2.12	Scenario 9 (S.09)	64
6.3	Discussion	65
6.3.1	Main observations	65
6.3.2	Threats to validity	66
6.4	Summary	67
7	Conclusions	70
7.1	Conclusions	70
7.2	Research Questions Revisited	71
7.3	Future Work	72
	References	74

List of Figures

1.1	Argo CD overview diagram	2
2.1	Growth of IoT devices compared to Non-IoT devices	7
2.2	Typical architecture of an IoT system	8
2.3	Traditional, virtualized and container deployment comparison	9
2.4	Wasmico architecture diagram	12
3.1	SLR filtering process diagram	17
3.2	Workflow of implemented solution by Paniagua <i>et. al</i>	19
3.3	Paniagua <i>et. al</i> solution system architecture	23
5.1	Ping request operation sequence diagram	41
5.2	Orchestration basic algorithm sequence diagram	44
5.3	Example limitation scenario for the basic orchestration algorithm	45
6.1	Metrics measured during Scenario 1 experiment	58
6.2	Scenario 2 experiment system diagram	59
6.3	Metrics measured during Scenario 2 experiment	60
6.4	Metrics measured during Scenario 3 experiment	61
6.5	Metrics measured during Scenario 4 experiment	62
6.6	Scenario 5 experiment system diagram	63
6.7	Metrics measured during Scenario 5 experiment	64
6.8	Scenario 6 experiment system diagram	65
6.9	Metrics measured during Scenario 6 experiment	66
6.10	Metrics measured during Scenario 7 experiment	67
6.11	Metrics measured during Scenario 8 experiment	68
6.12	Metrics measured during Scenario 9 experiment	69

List of Tables

3.1	Inclusion and exclusion criteria for the state of the art search results	16
3.2	SLR discovered solutions comparison	26
3.3	Grey literature discovered solutions comparison	27
6.1	Validation and evaluation test scenarios	57

Listings

5.1	Ping request example response	40
5.2	Get Heap Info request example response	41
5.3	Orchestration configuration file example	42
5.4	Terminal User Interface devices JSON file	51

List of Algorithms

1	CBBA	47
2	GetBestTask	48

Abbreviations and Symbols

AI	Artificial Intelligence
API	Application Programming Interface
(A)CBBA	(Asynchronous) Consensus-Based Bundle Algorithm
CD	Continuous Deployment
CLI	Command Line Interface
CPU	Central Processing Unit
DB	Database
DMG	Diminishing Marginal Gain
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IoT	Internet-of-Things
ML	Machine Learning
MQTT	Message Queue Telemetry Transport
OS	Operating System
RAM	Random Access Memory
REST	Representational State Transfer
RTOS	Real Time Operating System
SLR	Systematic Literature Review
SPI	Serial Peripheral Interface
SPIFFS	Serial Peripheral Interface Flash File System
SOA	Service-Oriented Architecture
TUI	Terminal User Interface
VM	Virtual Machine
WASM	WebAssembly

Chapter 1

Introduction

1.1 Context	1
1.2 Problem	3
1.3 Motivation	3
1.4 Goals	4
1.5 Document Structure	4

This chapter introduces the work that will be done in this dissertation. For that, we contextualize this thesis in Section 1.1. In Section 1.2, we succinctly describe the problem with current state of the art solutions and will describe this matter further in Chapter 4, p. 30. In Section 1.3, we show the motivations for producing the work in this thesis. In Section 1.4, we showcase the goals we aim to achieve at the end of the project and how we are proving their impact on developers. Finally, in Section 1.5, we show the general structure of the chapters in this dissertation.

1.1 Context

Over the last few years, significant progress has been made in the field of the Internet of Things (IoT) as its use and number of devices have been increasing significantly [46]. Applications based on IoT, such as smart cities or smart grid, have been progressively more present in our daily lives. These types of applications typically involve the use of numerous devices that work in unison to produce a final solution. However, this rising number of devices and tasks, the lack of manage tools, and the difficulties in automatically deploying jobs and moving them across devices pose a massive challenge that old technologies are not ready to tackle.

So, new technologies have emerged to overrule old software development and deployment trends. Among these, there have been advancements in containerization, orchestration, and continuous deployment (CD), whose popularity has increased in IoT as it helps design and manage this system more efficiently.

On the one hand, containerization software like Docker [15] has granted developers the ability to abstract their applications from the central systems in lightweight environments, contrary to pure virtualization using virtual machines. On the other hand, orchestration software like Kubernetes [32] for Docker containers has allowed developers to automate the management of their applications according to a defined architecture. The orchestrator guarantees that the system is always coherent and that resources are always optimally distributed between tasks. Finally, continuous delivery advancements like Argo CD [8] for Kubernetes have emerged to provide new tools to automate software deployment into testing or even production environments. The reality is that these three technologies have become increasingly popular together to form a more straightforward, streamlined, and automated development environment. In Figure 1.1, we can see the architecture of all of these technologies working together to automatically deploy merged changes onto a Kubernetes clusters while triggering other types of developer-defined events when new changes are uploaded in a repository online, in this case.

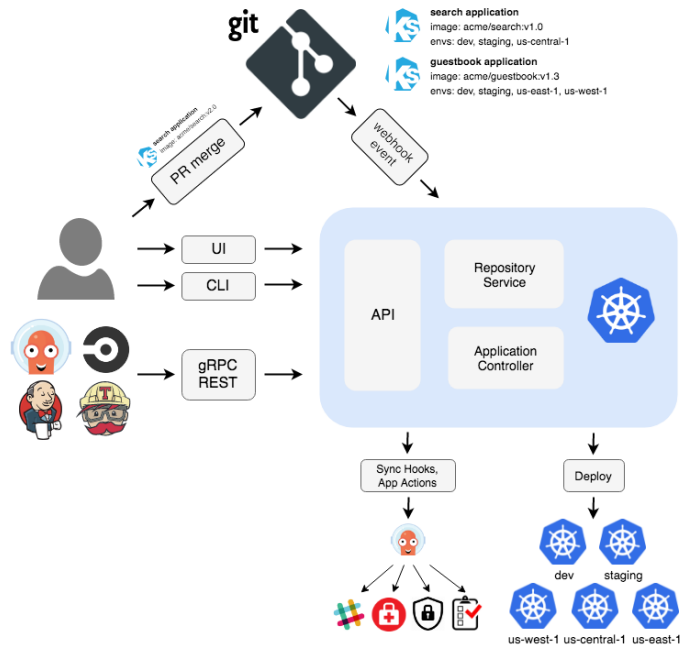


Figure 1.1: Argo CD overview diagram [9].

At the same time, hardware has also been continuously evolving, providing us with smaller computational devices, namely microcontrollers. These devices, which typically can fit in the palm of a hand, have limited resources and features but are highly specialized and power-efficient for the IoT. Still, only a few solutions target these platforms, which is becoming more of an issue nowadays.

1.2 Problem

As the microcontrollers used in IoT are generally resource constrained, Docker, Kubernetes, and Argo CD do not work on these as they were intended to function in machines that support a traditional OS (e.g. Linux). However, some containerization solutions have emerged to be employed in these kinds of devices such as Wasmico [42]. Moreover, there have also been some solutions when it comes to orchestration and DevOps, including CD operations, in these devices, namely Toit [50] and Goliath [21].

Wasmico does not yet have any means to orchestrate or automatically deploy software into the devices. Still, Wasmico using WebAssembly [54] language as a basis for the containerization gives it the versatility other tools do not since WebAssembly binary can be obtained through almost any programming language using external tools. In addition, the problem with the other two tools, Toit and Goliath, is that they either depend on some development pipeline and programming language or do not provide all the requirements an IoT product demands, including high availability, scalability, fault tolerance, security, and the ability to dynamically update and orchestrate the system at will. Not only that, but these solutions also always require a constant connection to the public internet, which in some scenarios might be impossible not only for the host system but also for the devices themselves due to physical constraints, for example.

We dive deeper into the problems of current state of the art solutions and our planned solution in Chapter 4, p. 30.

1.3 Motivation

Assuming the problems with the available solutions shared above, especially regarding orchestration and CD, and provided that Wasmico is a new and updated tool for containerizing tasks in constrained devices, it is only fair to iterate and add new features to it. A solution based on this architecture will give the developer multiple advantages, such as:

- **The programming language versatility** as the developer will be able to use any program language of their choice to develop their IoT applications.
- **An improved system for orchestration of multiple devices** by implementing mechanisms like fault-tolerance and scalability. Such features will create a system that is safer and easier to manage by the developer in the long run. Not only that, but these will also reduce the amount of manual errors and other issues that these systems face.
- **Continuing the work done in previous iterations** as we will be advancing for the next steps of previous research, specifically what was suggested by Pisco [37] in their dissertation. We will be performing this while addressing some open issues and gaps we find along the way with other tools.

In summary, our main motivation is to reduce manual labor from the developer in distributing tasks across their IoT devices while providing a systems that is more fault-proof and scalable for their application.

1.4 Goals

The main objective of this dissertation is to enhance Wasmico's capabilities especially when it comes to task orchestration between multiple devices and continuous deployment. This project is divided into two essential parts.

An orchestration system for Wasmico is planned to be developed. The primary mission of this orchestration system is to ease the discovery and deployment of tasks across multiple devices at the same time. While that is happening, the orchestrator is also be responsible for distributing resources across the deployed tasks.

After creating such component, we evaluate their impact and the benefits they bring to the table in IoT and DevOps. We do this by performing tests on the system using physical and virtual devices by measuring the execution times of tasks and how the system reorganizes itself according to unexpected changes.

1.5 Document Structure

The structure of this document is as follows:

- Chapter 1, p. 1, **Introduction**, introduces this dissertation, sharing information regarding its context, problem, motivation, and goals.
- Chapter 2, p. 6, **Background**, introduces some essential concepts that will be necessary during this thesis to better understand it.
- Chapter 3, p. 14, **State of the Art**, shares the state of the art solutions found through a Systematic Literature Review, in regards to the problem at stake, the orchestration of IoT applications in resource constrained devices, or microcontrollers.
- Chapter 4, p. 30, **Problem Statement**, states, based on the state of the art, the problems with those solutions and proposes a new solution to tackle those problems.
- Chapter 5, p. 37, **Solution Overview**, describes the implemented solution in detail. Particularly when it comes to the orchestration features implemented - fault-tolerance and auto-scaling.
- Chapter 6, p. 53, **Evaluation and Validation**, provides details on how we performed the evaluation and validation of our solution. It also presents its results across multiple scenarios.

- Chapter 7, p. 70, **Conclusions** succinctly concludes on all the previous chapters and goes back to the hypothesis. It also provides additional information on what future work could be done to better improve our work.

Chapter 2

Background

2.1 Internet-of-Things	6
2.2 Containerization and Orchestration	8
2.3 Wasmico	11
2.4 Summary	12

This chapter introduces topics that will be useful to comprehend the rest of this thesis. In Section 2.1, we present a definition for IoT, lay the architecture of its ecosystem and give some example applications. In Section 2.2, we explain containerization and orchestration as we delve deeper into the mechanisms of fault-tolerance and scalability in IoT ecosystems and its utility. Finally, in Section 2.3, we showcase Wasmico and its architecture and functionalities.

2.1 Internet-of-Things

The Internet-of-Things, or IoT, has become an incredibly captivating subject in our contemporary world as we increasingly depend on it. This growing significance is driven by the fact that most of our devices, whether computers or mobile phones, are in constant contact with the internet and inherently interact with each other. Although there are many definitions for what IoT is exactly, Madakam, Ramaswamy, and Tripathi refer to it as the interaction of “things”, living or not, over the “internet” [34]. However, they go even further and coin IoT as:

“An open and comprehensive network of intelligent objects that have the capacity to auto-organize, share information, data and resources, reacting and acting in face of situations and changes in the environment” (Madakam. Ramaswamy. & Tripathi. 2015)

Moreover, over the last decade, IoT systems have become increasingly more popular and, therefore, more present in our daily lives. As expected, due to this increasing interest, the number

of devices has also exponentially grown. This growth is expected to surpass the 30 billion mark by the end of 2030 [46]. On the same note, data suggest that about 75% of the devices worldwide will be IoT-related by 2025 [51] as it can be seen in Figure 2.1.

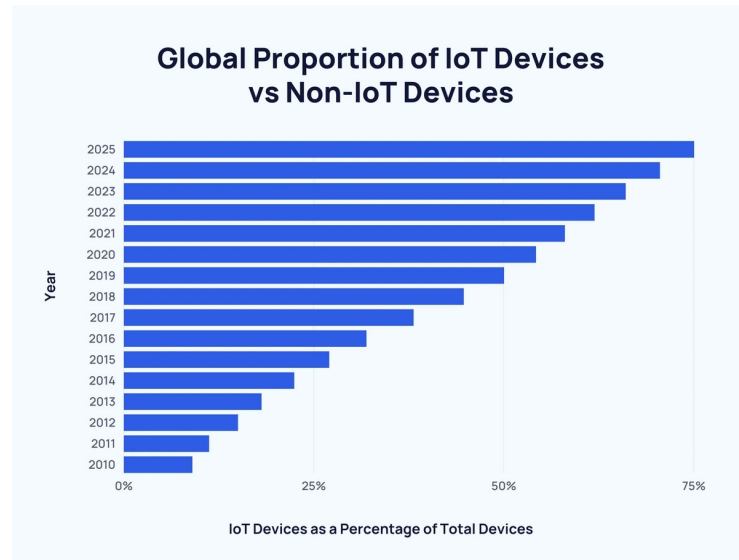


Figure 2.1: Growth of IoT devices compared to Non-IoT devices [51].

Nowadays, IoT has a plethora of different applications, but smart energy and smart cities are some of the most impactful in our lives as we currently live in these environments, for example.

For starters, smart meters are the first concept that jumps to mind when mentioning smart energy. These devices, present in almost every home nowadays, constantly report information regarding energy consumption in a home to a central server. The information provided via the internet by such devices is then processed and used for multiple reasons, whether to save energy or automatically process invoices for payment.

Secondly, smart cities have also been crucial to enhance people's lives. One example is intelligent public transportation systems, where, with ease, one can use their smartphones to access and track public vehicle information on the fly. These systems are powered by sensor devices installed within these conveyances that report such information to a central server using the internet in real-time.

When it comes to the architecture of such ecosystems, in recent years, the concepts of cloud, fog, and edge computing have been commonly mentioned to represent it despite this categorization still being controversial [13]. Cloud computing refers to high-resource and capability devices that process and save information from the lower layer, fog computing. These fog devices serve as a middleware layer that typically includes communication protocols and devices such as routers and gateways to ensure communication between the other two layers. Finally, on the edge, devices are commonly computationally constrained and provide minimal processing to the upper nodes [13].

In more concrete terms and according to Dias *et. al*, the IoT ecosystem architecture is commonly divided into three layers: perception, network, and application [13]. These layers have

different functions within the inner workings of an IoT product. The physical layer comprises devices, like microcontrollers, and its accessories let them be sensors or other input/output (I/O) devices. The network layer covers the protocols, technologies, typically wireless, and network devices used to communicate between these devices. Finally, the application layer typically has software for the specific cause, such as the ones mentioned previously, that normally uses a user interface that allows the user to view and even change different information regarding what is happening in the system. The diagram in Figure 2.2 draws these architecture:

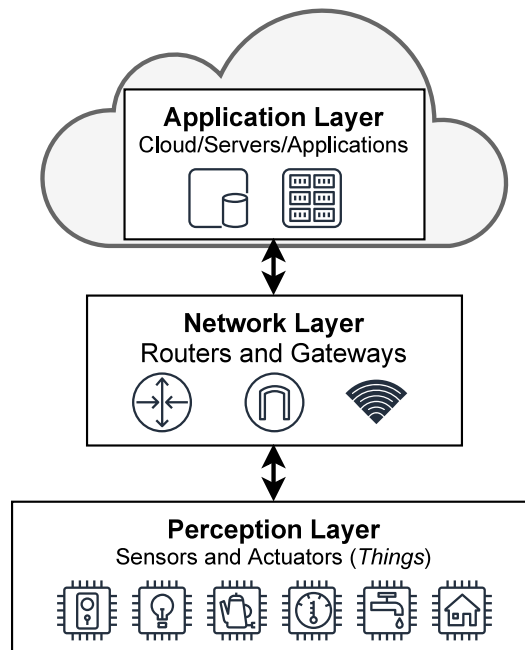


Figure 2.2: Typical architecture of an Internet of Things system [13].

2.2 Containerization and Orchestration

In order to deploy large applications over a significant number of devices, developers often rely on two paramount technologies - containerization and orchestration. Although they focus on different aspects, both these concepts are based on the idea that applications are abstracted from the user. Furthermore, the system behind orchestration is typically based on containerized applications, especially when it comes to lightweight applications with many components. The following subsections will delve deeper into these topics and how they have impacted how developers employ their applications worldwide, notably in IoT.

2.2.1 Containerization

Containerization is a software deployment process whose primary objective is the abstraction of programs from the host system. In order to do that, a container for each application is created.

This container incorporates all the necessary tools to build and run the software. This is done entirely independently from the host OS in its little separate environment without a full-sized OS behind it. The containerization of software encompasses multiple advantages, such as the ability to easily port the application from system to system or scale it through the same machine, creating multiple equally similar container instances [27].

In addition, containerization is originally based on the concept of virtualization. Virtualization is a technology that accurately simulates a physical system solely using software on the host OS, including all its components: the CPU, the RAM, the graphics card, or even I/O elements - a virtual machine (VM). Therefore, different hosts can virtualize identical systems, allowing both to run the same applications. Not only that, but these tools also allow the same host machine to have as many virtual machines as required for different applications. A hypervisor manages all of these virtual machines. The hypervisor guarantees that resources across the host system are correctly distributed over the different virtual machines depending on their necessities, optimizing the computational power flow. However, this type of virtualization suffers from the fact that it requires the emulation of a full-sized OS, something containerization does not, making it inherently heavier to run [28]. The diagram in Figure 2.3 better demonstrates the layer structure of these two deployment environments, virtualized and container:

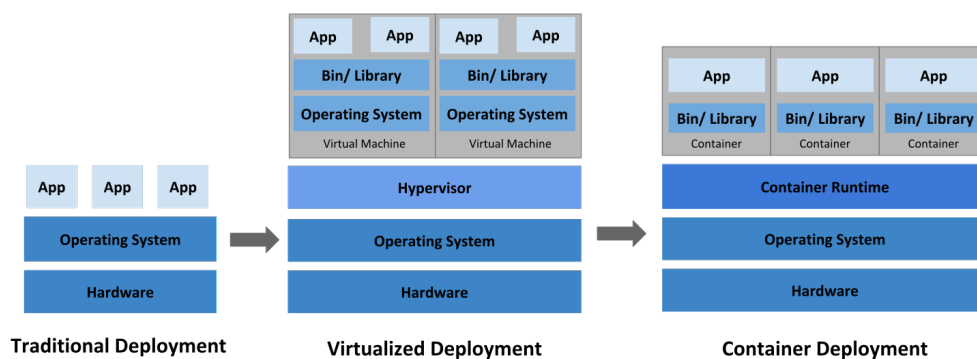


Figure 2.3: Comparison between different types of deployment and their layered structure - traditional, virtualized and container [32].

Nowadays, one of the most popular containerization tools is Docker [15]. Docker is an open-source, multi-purpose container engine runtime that generates software images from configuration files. The images generated from these configuration files contain all the software necessary to run the main application, including the code, system tools, library packages, and a container runtime instead of an OS. When these images are deployed into the Docker engine, containers are created. These containers are incredibly lightweight and have the flexibility to be duplicated, changed, paused, and transferred to other devices if necessary. Despite being as lightweight as they are, they are limited to host machines that can run full-fledged operating systems like Linux or Windows, meaning they do not support most embedded low-end systems.

In the realm of Internet-of-Things, since the devices typically used are resource-constrained and have many components, optimizing their utilization is essential. So, containerization is the

wisest choice as it is lighter than pure virtualization based in emulated VMs.

In the end, and considering everything mentioned above, containerizing programs and building an IoT ecosystem based on them makes the developer experience smoother and uncomplicated. This is because it allows the developer to streamline the deployment and management of their application on the fly [27].

2.2.2 Orchestration

Following up on the concept of containerization, there is often a link between it and orchestration. This relation exists because orchestration is typically connected to the automated management of large-scale applications that contain multiple microservices, generally in the form of containers. The orchestration system provides tools that automatically start, stop, and manage containers to avoid manual errors or failures. Moreover, these same containers can also be distributed across different systems without issue, as the orchestration platform can manage them as a whole [11].

In broad terms, container orchestration tools uses declarative programming as a way for the programmer to define the system's functionality in a configuration file. These instructions define how each container will behave and interact with each other. Therefore, the developer may define details regarding the system's internal network and the resources each container should share, for example. This way, when a new container is deployed into the existing pool, the system can automatically redistribute resources and keep operating in unison [11].

In general, orchestration systems generally offer features that ensure the performance and availability of their applications. Fault-tolerance is a task that ensures that multi-application systems do not fail as a whole if one of its parts also fails. To achieve such a feat, it requires the implementation of strategies that remove *single points of failure*. Two of these strategies are redundancy and replication [18].

Redundancy is the instantiation of a group of duplicate containers or devices ready to take action if the main one ever produces a catastrophic failure. Replication, on the other hand, is more complex than that, as these duplicates function as the one cloned. However, the orchestrator periodically confirms if all these containers produce identical behaviour. If they do not, a democratic procedure is performed to assess which systems are at fault. These systems will later be prosecuted according to the specifications defined by the developer. Both of these mechanisms can be implemented on both the orchestrator or the constrained devices levels.

Another feature is scalability, which refers to the mechanism that allows an application to scale up and down according to its needs. This means that if a workload becomes too much for a selected number of containers, the orchestrator can instantiate more to fill the gaps. Moreover, the opposite is also possible, so if the load is underwhelming, the system can kill some of the containers and save precious resources that other applications might use.

Currently, one of the most powerful and popular orchestration software is Kubernetes [32]. Kubernetes is an open-source platform used to manage containerized workloads across various systems. It comprises tools that mirror all the previously mentioned orchestration features. For

example, as referred on their website, the Kubernetes orchestrator is capable of performing self-healing operations where it can automatically restart and reallocate resources for containers that fail health checks (fault-tolerance).

Finally, the orchestration of applications has become increasingly popular, and its use has also spread to the IoT field. It provides numerous tools that deliver invaluable advantages such as the ability to analyze the performance of an application and perform health checks; better performance and availability as the orchestrator has the tools to automatically restart, replace, or move a container to another compatible system if a failure occurs; and scalability as the number of containers can be scaled up or down according to all the variables and parameters the system is subject to. All of this reduces the human resources and the repetitive tasks required to manage these complex systems, thus reducing overall costs [11].

2.3 Wasmico

Most modern microcontroller devices support lightweight operating systems and frameworks that help build applications. For example, the Arduino [4] and the ESP-IDF [48] platforms have their own independent frameworks for their types of devices. However, the market offers solutions that support different devices types, including the ones mentioned previously, such as MicroPython [35] and JerryScript [29]. MicroPython is an optimized version of Python3 for microcontrollers. It provides many of the comforts native Python offers without being too heavy for these constrained devices. JerryScript, on the other hand, is a javascript engine intended for the IoT. It gives the user easy access to the I/O while being incredibly lightweight, especially for resource-constrained microcontrollers.

Another attractive solution would be WebAssembly [54] (WASM) for its lightness, popularity in IoT, and especially for its versatility as it indirectly supports multiple languages. This is because native programs from other languages, such as Java, Rust, Javascript, and C/C++, to name a few, can be easily compiled into WebAssembly using existing tools. The market offers multiple runtimes and interpreters, but Wasm3 [52] is, by far, the most popular.

With all of this in mind, Wasmico [40, 42], available online¹, was born. It is an open-source containerization tool to deploy and manage WASM-based containers in low-end microcontrollers. The programs to be run by the Wasm3 interpreter on each device can be developed in any programming language and then converted, using existing software, to lightweight WASM binaries, as previously mentioned. Afterward, these generated binaries can be uploaded to the device's filesystem (SPIFFS) using the provided command line interface (CLI).

Effectively, Wasmico allows the user to perform other actions besides just uploading and running containers. They can also stop, delete, pause, and unpause them on the fly through the same CLI. Internally, all of these commands, including the WASM binary upload, are sent through an HTTP connection to a web server running on each of the microcontrollers. The microcontrollers also run the FreeRTOS real-time operating system [20] whose responsibility is to schedule the

¹<https://github.com/SIGNEXT/wasmico-ng>

containers' tasks and the HTTP server to ensure they are run concurrently, assuming some pre-defined time constraints. In Figure 2.4, we can see a summary of this architecture and its inner workings.

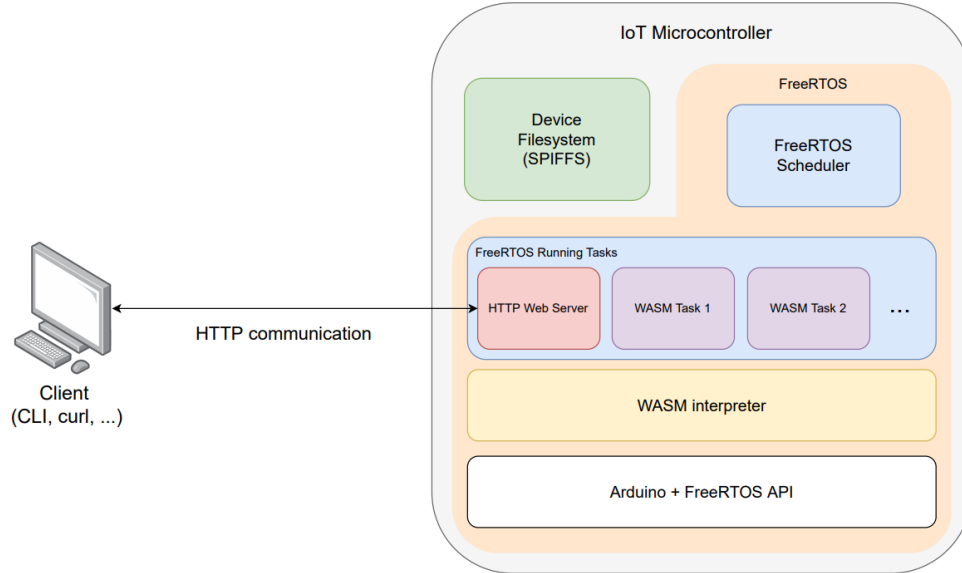


Figure 2.4: Wasmico architecture diagram [40].

However, Pisco *et al.* [37] expanded on these features in their thesis, namely when it came to container migration and user experience. Regarding container migration, they introduced state saving and restoring operations. This way, it is possible to transfer a task's current state from one microcontroller to another with similar capabilities running the same task. Moreover, to improve the user experience, they presented a terminal user interface (TUI), where the user can interact with any of the microcontrollers on the network in a more user-friendly way compared to the previously developed CLI. This improvement was tested via an empirical study with potential users of the system. Finally, they also introduced some improvements in the general functionality and efficiency of the tool.

2.4 Summary

Throughout this chapter, we introduced multiple topics that will be important for clearly understanding the rest of this dissertation.

In the first section, Section 2.1, p. 6, we introduced the topic of IoT and how it has been growing in today's world. Furthermore, we also shared some of the most prevalent applications in the area and conveyed their influence on our lives nowadays. Finally, we went over the architecture of a typical IoT application and how that ecosystem works internally.

In the second section, Section 2.2, p. 8, we briefly explained the concepts of containerization and orchestration. When it comes to containerization, we clarified its close relation to virtualization and how it is computationally cheaper to execute, especially in resource-constrained

machines. What is more, we also mentioned how it is related to IoT and how it can be helpful compared to other approaches. Finally, regarding orchestration, we explained that when correctly incorporated, mechanisms such as fault-tolerance and scalability significantly help create a highly available and operational system, particularly for the IoT, which is crucial to the applications we mentioned in the first section, for example.

In the third and final section, Section 2.3, p. 11, we summarily described where the Wasmico tool came from and what its internal architecture looks like. Besides, we conveyed some of its functionalities and how they have been improved to be up to the state of the art over the latest works.

Chapter 3

State of the Art

3.1	Systematic Literature Review	14
3.2	Grey Literature	22
3.3	Results Analysis	25
3.4	Conclusion	28

During this chapter, we will explore state of the art solutions that cover the orchestration of applications in resource-constrained devices in IoT. In Section 3.1.1, we will cover the Systematic Literature Review approach methodology taken in research for these solutions. In Section 3.1.6, we will briefly cover each one of the solutions found, focusing on the orchestration features. Still, regarding the Systematic Literature Review, we will explore some extended results in Section 3.1.7. In Section 3.2, we will review some solutions not documented in the literature as grey literature. In Section 3.3, we will analyze the results obtained according to some parameters. Finally, in Section 3.4, we will conclude the analysis done in the previous section regarding the state of the art found as a whole.

3.1 Systematic Literature Review

In order to capture papers and other relevant projects in the state of the art pertinent to this thesis, we performed a *Systematic Literature Review* (SLR). The SLR is a complete and comprehensive study of the existing literature that fits the theme of a research. In the following sections, we will introduce this process and how it was executed to explore the matter under study. Furthermore, we will share our results and study the extension we performed on them according to some criteria.

3.1.1 Methodology

In this review, research questions, which later evolve into search queries, are the basis of our study. These queries are then used on chosen online databases that contain a gigantic amount of research

papers and projects. Finally, the returned results are pithily analyzed according to their contents and filtered. These results are finally studied and a conclusion is drawn on them.

3.1.2 Research Questions

For our work, we formulated the following research questions to help us craft our queries to potentially find the gaps in the literature regarding the matter at study.

SRQ1 *What are the existing solutions for orchestrating applications in low-end microcontroller devices in IoT?* This first research question aims to find all the solutions to attack our problem. More specifically, this research question aims to find device orchestration solutions that are directly intended to be used for IoT applications. Furthermore, we also intend to explore how some orchestration concepts/features are implemented, such as fault tolerance and scalability, though this is secondary.

SRQ2 *From the results of SRQ1, which of these solutions allow code updates over the air (OTA)?* Finally, this last research question makes our search focus on solutions that allow our applications to be updated/reprogrammed wirelessly without flashing the devices. This approach, also called reprogramming on the fly, will be fundamental to explore to complete our study.

We decided not to include a question regarding the usage of a WASM runtime environment for the applications as this would lock our research on one particular technology, and WASM was not our focus here.

3.1.3 Search Queries

From the previously defined research questions, there are a few keywords we can deduct that will be relevant in reaching the best possible results: *Internet-of-Things*, *Microcontroller*, *Device* and *Orchestration*.

So, using this keywords, we formulated the following complete search query (orchestration AND (iot OR "internet-of-things" OR "internet of things") AND devices AND microcontrollers).

With this query, we are able to explore all the existing literature regarding orchestration of internet of things applications on microcontrollers.

3.1.4 Online Databases

For this research and specifically for the Systematic Literature Review, we selected three major digital databases: ACM Digital Library¹, IEEE Xplore² and Web of Science³. We chose these three specifically since they are the most appropriate for the matter we will be researching, and they are quite complete with the amount of papers they contain.

¹<https://www.acm.org/publications/digital-library>

²<https://ieeexplore.ieee.org/>

³<https://www.webofscience.com/wos/woscc/advanced-search>

3.1.5 Results Filtering

After gathering all the results from the selected databases using the previously defined query, we had to filter them. To do so, we defined a list of inclusion and exclusion criteria in Table 3.1. To apply this table, we will briefly study the contents of the papers obtained, for example, their title, abstract, and introduction, and consider their inclusion and exclusion in our study.

I/E	ID	Description
Inclusion	IC1	The work goes over the topic of task orchestration in an Internet-of-Things context.
	IC2	The work allows updates over the air (OTA).
	IC3	The work mentions orchestration mechanisms namely fault-tolerance and/or scalability.
	IC4	The work uses a WebAssembly runtime environment for the tasks.
Exclusion	EC1	The work is not in English.
	EC2	The work is duplicated.
	EC3	The work is based on edge devices that run Unix-like operating system. Linux e.g.
	EC4	The work does not use physical microcontrollers, reverting to simulation, for example.
	EC5	The work does not exist in a readable format, PDF e.g.

Table 3.1: Criteria for inclusion and exclusions of the results obtained through the previously defined query on the selected databases.

In the previous table, we determined a total of four inclusion and five exclusion criteria for our research. For starters, IC1 and IC2 were designed to help include all the papers directly related to the matter in study, including orchestration in Internet-of-Things and application updates over the air (OTA). The other two inclusion criteria, IC3 and IC4, dive deeper into the topic we are exploring. However, we will primarily use these criteria to sort the results found by their relevance since these are just desirable details we seek.

On the other side of the spectrum, exclusion criteria EC1, EC2 and EC5 were employed to remove duplicated work and work that we cannot read - not written in English or not in a readable format, PDF e.g. Then, EC3 removes all the papers where the orchestration system is based on edge devices that can run full-size operating systems, particularly Unix-like, such as Linux. The last criterion, EC4, removes all the works that do not use physical microcontrollers for any of their testing, resorting to emulation, for example. The usage of physical devices is significant since not only will our work revolve around them, but also, they fail in ways that are hard to simulate using only software.

Finally, to employ these criteria and gather the final results, we followed the following steps:

1. **Complete database search:** Using the databases we selected with the aforementioned query, we extracted all the results found.

2. **Initial filtering:** Removed duplicates, non-readable documents, and the ones not in English (*EC2*, *EC5* and *EC1*) from the extracted results.
3. **Metadata filtering:** By analyzing the metadata of each paper, we extracted those that were related to the field of orchestration in an IoT context (*IC1*).
4. **Introduction and Conclusion filtering:** We used the same process as the last step but applied it to the Introduction and Conclusion of each work.
5. **Full-text filtering:** Now, we expanded our analysis to the full text of the documents, removing the ones that use devices that can run full-size OS, the ones that are not using physical devices (*EC3* and *EC4*). In this step, we also made sure to include only the papers that mentioned over the air updates as one of the features of their solution (*IC1* and *IC2*).
6. **Results expansion:** In this step, we expanded the results found by adding articles that are relevant but not captured by the query and from expert feedback.
7. **Sorting by relevance:** Finally, we will sort the remaining papers by relevance. To do this, we will be taking into account if their solution mentions orchestration mechanisms, namely fault tolerance or/and scalability or/and uses WebAssembly as a runtime for the tasks in the system (*IC3* and *IC4*).

With this process, after the first step, we extracted 638 entries, and, in the end, our filtering led to eight papers that we will get more into in the following two sections, Section 3.1.6 and Section 3.1.7. The breakdown of the whole filtering process can also be observed in Figure 3.1.

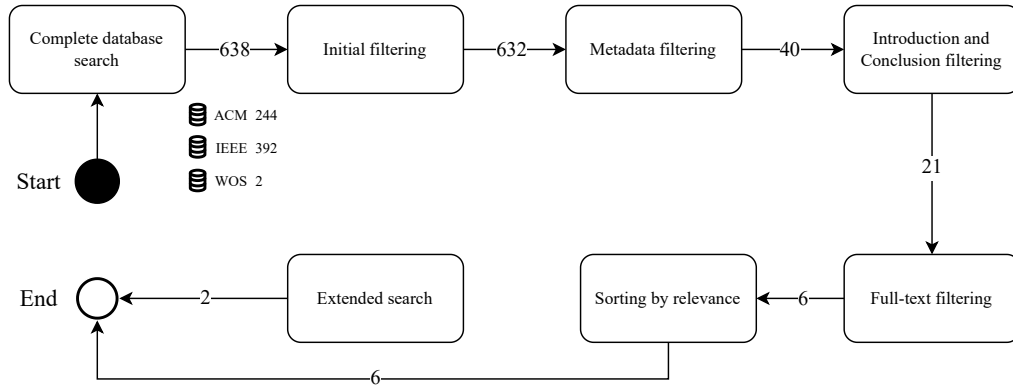


Figure 3.1: Systematic Literature Review filtering process diagram.

3.1.6 Main Results

In this section, we will delve into the six main results of our SLR by analyzing their papers and other relevant scientific documents. These are as follows:

Anatolijs et. al [2] describe how IoT and a Service-Oriented Architecture (SOA) can be applied to Municipal Systems Infrastructure Control to develop more reliable and accurate systems

for tracking information around a city. To achieve this, they implemented the Message Queue Telemetry Transport (MQTT), a lightweight pub-sub IoT protocol, to perform the communication between sensor devices across the city. These connections enable the system to be more closely connected with little to no latency (edge-to-edge) and open the doors to a very reliably scaled ecosystem. This MQTT protocol implementation is also the basis of the orchestration system integrated into Node-RED [24], which uses visual programming to represent data flows and behaviors between tasks while allowing the end user to deploy and request data on demand.

With that in mind, to obtain the data from the edge devices, three steps are necessary: subscribe to the wanted topic; edge devices form two distinct payloads, one with the data serial number and the device id and another with the data block itself to be decoded; finally two MQTT messages (double output) are sent where each contains different devices measurements and two different payloads. All of this data flow and orchestration is managed by Node-RED.

Paniagua et. al [36] describe a solution that aims to remove the latency from cloud-edge task invocation by passing this task to the edges themselves. This reduction in latency enables the creation of more extensive, less power-hungry device networks. The authors highlight that such a feat is fundamental in IoT ecosystems nowadays as these restrictions are increasingly in demand.

In order to fulfill this functionality, it first uses a typical orchestrator to dispatch a task requested by one device to another. Then, when the second device replies, it sends the results back to the orchestrator. The orchestrator then processes this information and informs the primary device that the start task request was made through a specific endpoint on the second device. Now, the next time device one requests the same task to be run, it no longer needs to communicate with the orchestrator since it can directly connect to device two through that endpoint and start the task itself. In Figure 3.2, it is possible to see the workflow and device connections of this execution, where A, in blue, represents the first phase and B, in orange, represents the second phase.

In their testing, they used a resource-constrained ARM Cortex-M4 device running the RIOT operating system [41] with a simple communication interface. Their results show that this approach reduced latency by about 86%, which was satisfactory for their objective. They also share that this approach, despite being dependent on many factors, such as the performance of the wireless communication system, early theories suggest that using it will entice the scalability in large-scale IoT systems.

Baccelli et. al [6] report, in their paper, a prototype of an orchestration system for low-end IoT devices. For this prototype, they used cheap IoT microcontrollers running the lightweight RIOT operating system [41]. As for the framework of choice, they used JerryScript [29], a light Javascript engine. As for the cloud, they use a cloud-based IDE that can convert simple

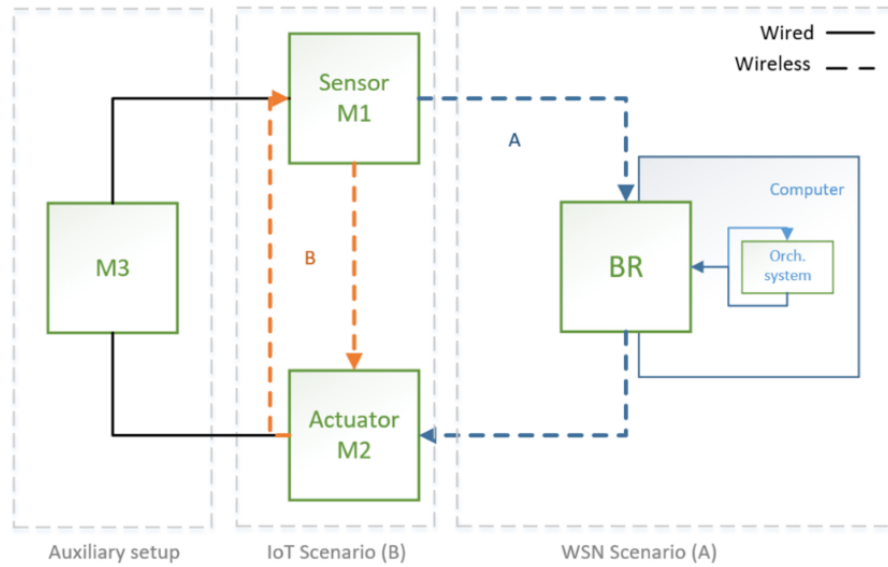


Figure 3.2: Workflow of implemented solution by Paniagua *et. al* [36].

graphical programming programs created in it to Javascript. Finally, they used a network gateway device with a simple radio transmitter/receiver for the communication interface.

For their testing, the authors created a simple application and pushed the result to be installed into the edge device. They performed this action multiple times to tune the application, which updates it without flashing the whole memory. The application updates mentioned are possible since JerryScript creates little sandbox environments that theoretically work like containers. Their results show that partially updating the system like this uses less memory, 1Kb vs 60Kb, making it considerably faster if we consider booting and data transmission times.

Finally, at the end of the paper, Baccelli *et al.* note that partial patching of containers would further improve the savings in updating applications on the edge if changes are minimal. Techniques like dynamic loading of binary modules or differential binary patching would be the choice to develop such an idea.

Tiny-MLOps [3] is a solution that aims to orchestrate the execution of machine learning (ML) applications on resource-constrained far edge devices. In this paper, the authors explain that their solution is just an adapted version of the MLOps framework but, this time, oriented to devices with limited resources. This altered framework comprises the following steps:

1. *Plan*, where the target platform categorizes the devices to run tasks that their hardware is better designed for;
2. *Create*, where the devices partially run the ML algorithm and take advantage of locality to produce more accurate results;
3. *Adapt and Optimize*, where the ML model generated is optimized to be used on these constrained devices;

4. *Verify*, where the model goes through more checks to ensure it is suitable for inference;
5. *Package*, where the model is finally compiled and processed for the desirable target platform;
6. *Release*, where the compiled model is installed on the target platform in the form of loading it into memory;
7. *Configure*, where the model is tuned to adapt to the environment dynamically. These configuration values depend and are tuned based on the model's performance and how it compares to others on the model pool;
8. *Monitor*, where telemetry data is pulled from the model performance in real time to check for any errors or changes. This phase can also trigger another iteration of the process if required to enhance the models already deployed.

As for testing, this framework was used as the basis to deploy and test an unsupervised anomaly detection model over a device on edge inserted into an extreme condition scenario where its accessibility is minimal. It is important to note that the original paper provides more details about this specific implementation, which we will not delve into here. According to the authors, these tests with physical devices revealed promising results regarding metrics like deployment, loading, and inference latency. However, it is also revealed that this solution still has some limitations, like deployment time, due to the uploading of unoptimized models into the slow embedded devices using primitive wireless communication modules.

Guerin et. al [22] describe a framework that allows the orchestration of cheap microcontrollers to form complex services using the Lua scripting language [33] and virtual machine. Their architecture does not contain a central device as the orchestrator but instead mobile devices that actively discover the embedded platforms and perform actions on them.

Effectively, deploying tasks to the devices takes the following three steps. First, mobile devices discover surrounding embedded systems by scanning the environment and finding the signals they are broadcasting. Once they find these devices, they process their data, including information regarding their capabilities and current status. The mobile orchestrator will then use this information to decide where and whether to deploy some requested tasks. When the discovery phase has been completed, and the orchestrator has decided what to do with the tasks at hand, a single and temporary connection is made to the embedded device, using Wi-Fi, for example, to install the script. Finally, the edge device, which uses FreeRTOS as the base OS and the Lua VM to run the applications, executes the script the orchestrator sent if it meets all the appropriate conditions and terminates the wireless connection.

Regarding testing, the authors used ten ESP32 [48] devices and the computer as the mobile orchestrator. Throughout their testing, they realized that deploying and executing tasks like this is considerably faster than installing new versions by flashing the whole memory. However, they also reached the conclusion that most of the delays caused by this framework are

due to connectivity latency and other overheads, something that can be further optimized in the future.

GeneSIS [17] is a framework for continuous orchestration and deployment of smart systems applications. This tool was built to cope with the heterogeneity of today's IoT ecosystem devices on edge. GeneSIS currently supports three types of deployable artifacts to archive this diverseness compatibility with many devices: blackbox, ThingML [49], and Node-RED [24]. This framework includes a modeling language that defines the architecture of the system and an engine that implements this architecture in the form of orchestration and dynamic deployment.

The modeling language provided by GeneSIS is conceptually straightforward but very easily expandable. It comprises building blocks that can be connected to define the infrastructure of the smart system. These building blocks and their functions are detailed on their paper. With this system definition, the resulting system is responsible for deploying the software components, assuring they communicate with each other, orchestrating resources between them, and tracking their status and other telemetry information.

Moreover, the execution engine implements a typical Models@Run-time pattern. This design allows the engine to adapt the system automatically when changes occur. New deployments, crashes, and updates over the air, among others, are some of the changes that trigger such events. The adaptation engine simply changes the necessary components to optimally reach the final objective.

3.1.7 Extended Search

In the previous section, we went over the results of our SLR. However, more exploration of the selected databases revealed that one particular article is still interesting to showcase here despite our query not capturing it. Besides that, we obtained a second relevant document for our research through expert feedback on the matter of study.

With all of this in mind, the two articles are as follows:

ANIARA Project [30] is a solution that aims to bring computational capacity to edge devices for technologies like smart manufacturing and smart cities to reduce latency, especially when AI and ML are in play. To achieve this, they developed an infrastructure based on lightweight WebAssembly [54] containers and a fast database, RonDB [23]. Not only that, but the use of cloud-edge orchestration on these devices ensures that the needs of these types of applications are satisfied.

This orchestration is archived via network slicing in their infrastructure, allowing for the provisioning of different services over the same 5G network in totally independent connections. For each of these slices, resources are allocated, as well as backup resources in case of link and node failures. In their paper, these resources are not just computational but also

connectional since, according to the authors, orchestrating them separately would lead to impractical solutions.

These backup resource allocation schemes for orchestration must be developed to ensure that resources are distributed optimally between all the slices to reduce the overhaul required resources. While the paper goes over some of the explored solutions found in the literature, they settled for a heuristic-based shared protection algorithm that optimizes the sharing of these backup resources between devices. Finally, they show that their approach is better at maintaining service quality than other solutions through a simulation, although other tests were also performed on inexpensive microcontrollers.

Silva et. al [45] describe an application orchestration solution for resource-constrained devices based on the visual programming language Node-RED [24]. To do this, they performed modifications to Node-RED's engine and produced a custom firmware for the low-end devices since most cannot run Node-RED's workflows natively. This firmware is based on MicroPython [35], an undersized version of Python that can be used on low-end microcontrollers to create lightweight environments.

In order to perform discovery and connection of new devices, two steps are taken. First, the devices announce their presence through a MQTT topic. The orchestrator captures this information, which includes data regarding the device's capabilities, and saves it on a Registry Node. Second, the orchestrator forms a connection to the HTTP server on board of the microcontroller to send any MicroPython scripts that the end user requests.

The orchestrator is then responsible for deploying requested tasks to available nodes capable of performing such jobs. Also, each node carries a set of predicates and priorities that command their hard and soft constraints, respectively. This node allocation is determined by a greedy algorithm that considers all of these previous factors. The devices themselves have fail-safe mechanisms in place that trigger if any internal fatal error occurs. If such an event were to happen, they would quickly restart all their nodes and reinstate the HTTP connection to the orchestrator. The architecture previously described can be found in Figure 3.3.

Finally, in their paper, they performed extensive testing of the developed tool in thirteen scenarios related to home automation within two experimental environments. Based on this, the authors concluded that their solution is viable as an orchestration framework for the IoT since it can orchestrate devices and re-orchestrate them dynamically as the environment changes while using low-end devices.

3.2 Grey Literature

After collecting all the results from the previously described SLR process, we decided to expand from that by doing manual searches on Google⁴, which enabled us to explore more sources online.

⁴<https://www.google.com/>

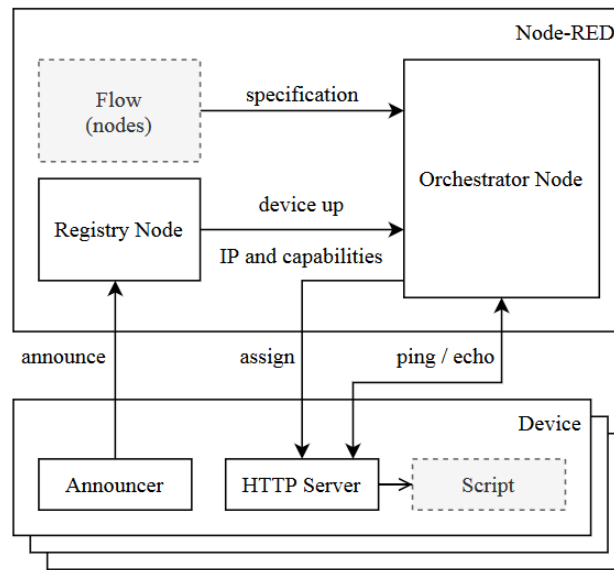


Figure 3.3: Paniagua *et. al* [45] solution system architecture.

The results obtained are valued as grey literature since they represent projects and products that are not scientifically documented in literature.

3.2.1 Results

The results obtained through such means are as follows:

Goliath [21] is a cloud platform that allows developers to manage their embedded devices, namely those in the field of the IoT. It supports different types of devices, scilicet ESP32 ESP-IDF [48] and those compatible with the Zephyr real-time OS [38] based on FreeRTOS [20].

Goliath provides many tools through its web console (GUI), command line, and REST API. These tools allow the user to tailor their desired solution as it is possible to add, delete, edit, view, and interact with the connected devices. These interactions can range from simply querying data from one of the devices to providing over the air updates. The OTA updates feature is based on three concepts: artifacts containing binary images of updates to the devices, releases containing multiple of these artifacts that are sent to the devices, and blueprints with a map of the resource allocation of the receiving system, hence, it knows where and how to allocate these packages. Not only this, but Goliath also delivers tools to record device data and device telemetry. With said data, it is possible to create automated workflows and processes. Besides, this data is saved into a lightweight, simple database that can be easily queried.

Toit [1], like Goliath, is a tool to manage containers in microcontrollers. However, Toit is exclusively available to ESP32 devices. Despite this limitation, they pride themselves on being a fast and safe development framework as their programming language, Toit language [50], is high-level, memory-safe, developer-friendly, and secure. Meanwhile, the microcontrollers'

virtual machine and container firmware is light and battery-saving oriented, which is quite meaningful for devices this size.

As a container management toolbox, Toit contains tools to track devices, access logs, and update and install applications on the registered devices. Moreover, these functions can be integrated into a continuous delivery pipeline to update the devices automatically on every new commit. All of this is, of course, according to a specific architecture previously defined by the developer on system initiation.

ESPHome [7] is a tool to manage home automation applications on microcontrollers using configuration files. It supports five microcontroller platforms, namely the ESP8266, ESP32, RP2040, BK72xx, and RTL87xx. The infrastructure of the applications is defined on a Yaml file according to their specification, although it is possible to embed C++ external code as well.

This tool currently supports multiple types of components, including regular sensors, binary sensors, light components, switch components, button components, fan components, display components, and speaker components, to name a few. These components are already pre-defined and pre-programmed internally in C++, so they just need to have their parameters configured on the YAML file for them to work. Moreover, the core components provide the developer with mechanisms to set up communication protocols between devices, a logger, an HTTP server, and over the air updates. In particular, the over the air updates feature can be protected by a secure password for added security. Finally, since the project is open source, many libraries are available that integrate and add additional features to the currently available ones.

Azure IoT Hub [5] is a cloud platform solution for managing IoT applications across numerous devices. However, compared to the two previous commercial products, the ones supported range from fully-fledged computers to tiny little sensor microcontrollers.

Microsoft's Azure IoT Hub enables IoT systems to be reliable and secure through their communication channels. Through these channels, messages between the cloud and the edges and information about the state and telemetry of each device are passed securely. Not only this, but it also provides communication mechanisms between edge to edge. Most importantly, these channels support OTA updates, enabling high-scale applications in the IoT ecosystem to be rapidly updated. In order to archive this functionality, a device update agent is installed in each device, where every time they receive an update command to either download, install, or apply, they execute that function accordingly.

Also, since Microsoft owns this service, it integrates nicely with its ecosystem, providing many additional features, such as the ability to build hybrid and dynamic applications with Azure IoT Edge.

AWS IoT Greengrass [25] is an IoT cloud platform and edge runtime that is used for building, deploying and managing IoT application on edge devices. The devices officially supported

are either Linux, Windows based or use Docker [15] containers. However, they note that devices under other platforms might also work, like the ESP32 [26].

Greengrass has many features, including local device processing, scale management, and over the air updates. With local device processing, devices can gather information and process it on the edge before aggregating it between edges and sending it as a whole to the cloud. Greengrass's scale management allows components to be easily deployed and managed on millions of devices. It also allows the organization of these devices into different groups that can be handled simultaneously. Finally over the air updates for new firmware or components are readily accessible to perform using their web interfaces.

Again, as Greengrass is part of a large company like Amazon, it integrates well with its ecosystem, providing many additional features. For example, these include using AWS's Lambda Function on the devices and managing them through Greengrass.

3.3 Results Analysis

In this section, we will further explore our results by comparing them against each other across various parameters. We will also draw some observations on both the SLR and the grey literature results before concluding on them altogether on the next section, section 3.4.

3.3.1 Parameters

In order to compare and evaluate each of the tools explored in the previous sections, we decided to define a list of parameters and their possible values, which are the following:

1. **Application:** Some of these tools were built for some specific use cases. *Possible values: Name of the application, Multiple if the solution is general purpose, or N/A if no information is available.*
2. **Constrained OS:** Each of these tools uses a different operating system on their constrained devices. *Possible values: Name of the OS, Multiple if multiple OS are supported, or N/A if no information is available.*
3. **Constrained language:** Each of these tools uses a different programming language for each of their constrained devices. *Possible value: Name of the language, Multiple if multiple are supported, or N/A if no information is available.*
4. **Software as a Service (SaaS):** Each of these tools is provided as either a Software as a Service or as a self managed software. This information is especially relevant since it reflects on the internet dependency of the system. *Possible value: ✓ or ✗.*
5. **Scalability (S):** Each of these tools have different abilities to reliably and optimally scale into larger systems. We will evaluate this metric solely based on the testing and statements

provided. *Possible value: Number of devices supported in shown tests, if available, Low or High if this detail is not available concretely, or N/A if no information is available.*

6. **Fault-tolerance (FT):** Each of these tools have different abilities and mechanisms to reliably and optimally recover from a fatal error state. We will evaluate this metric solely based on the testing and statements provided. *Multiple possible value(s): OR (Orchestrator Redundancy), DR (Device Redundancy), TR (Task Reallocation), or N/A if no information is available.*

7. **Telemetry (T):** Each of these tools can have monitoring and telemetry services available. *Possible value: ✓ or ✗.*

3.3.2 Systematic Literature Review

For each of the tools shown on our Systematic Literature Review, in Section 3.1, we evaluated each one of the parameters detailed above, in Section 3.3.1, and obtained the following results as shown in Table 3.2.

Solution	Application	Constrained OS	Constrained language	SaaS	S	FT	T
Anatolijs <i>et. al</i> [2]	Smart city	N/A	N/A	✗	N/A	N/A	✓
Paniagua <i>et. al</i> [36]	Multiple	RIOT OS	C	✗	High ¹	N/A	✓
Bacelli <i>et. al</i> [6]	Multiple	RIOT OS	Jerryscript	✗	N/A	N/A	✗
Tiny-MLOps [3]	ML	N/A	MicroPython	✗	N/A	TR	✓
Guerin <i>et. al</i> [22]	Multiple	FreeRTOS	Lua	✗	10	N/A	✗
GeneSIS [17]	Multiple	Multiple	Multiple	✗	High ¹	OR, DR, TR ²	✓
ANIARA Project [30]	Smart city	N/A	WebAssembly	✗	High ¹	DR	✓
Silva <i>et. al</i> [45]	Multiple ³	N/A	MicroPython	✗	50	DR, TR	✗

Table 3.2: Comparison between all the solutions discovered in our systematic literature review investigation.

¹ No information regarding the number of devices it can scale to is explicitly available. However, a general analysis of the document leads us to conclude how heavily the system can be scaled.

² The certainty concerning this information is relatively low.

³ Solution was tested on an home automation scenario.

Regarding our Systematic Literature findings, most of the solutions found provide details on the technologies used for each part of the system. This effectively limits how broad the solutions produced are. However, the final products tend to be more condensed and more uncomplicated to set up while providing a wide range of functionality for each specific application with ease.

3.3.3 Grey Literature

We performed the same analysis on the grey literature results on Section 3.2 and obtained the results as shown in Table 3.3.

On the other hand, in our grey literature, we find that most of the solutions found are general purpose and, therefore, support different OS, devices, and programming languages. However, as they are not confined to a standard and are versatile products, they usually require extra configuration steps to reach solutions to cater to each need.

Solution	Application	Constrained OS	Constrained language	SaaS	S	FT	T
Goliath [21]	Multiple	Multiple ⁴	Multiple	✓	High ¹	N/A	✓
Toit [1]	Multiple	Multiple	Toit Language	✓	High ¹³	TR	✗
ESPHome [7]	Smart Home	N/A	Yaml ⁵	✗	N/A	N/A	✗
Azure IoT Hub [5]	Multiple	Multiple	Multiple	✓	High ¹	DR, TR ²	✓
AWS IoT Greengrass [25]	Multiple	Multiple	Multiple	✓	High ¹	DR, TR ²	✓

Table 3.3: Comparison between all the solutions discovered in our grey literature investigation.

¹ No information regarding the number of devices it can scale to is explicitly available. However, a general analysis of the solutions and its documentation leads us to conclude how heavily the system can be scaled.

² These features are not available out-of-the-box but can be implemented with the available integrations.

³ Using Artemis, a tool that is integrated into the Toit ecosystem.

⁴ Primary support to Zephyr [38] but compatible with other real-time OSs.

⁵ Supports C++ under the use of their API - the ESPHome API.

3.3.4 Overview

With both tables considered, we analyzed each column individually and concluded on them, as can be seen by the following:

Application - Most of the solutions, 9 out of 13, especially those found in grey literature, were not built to fit one particular application. However, the solutions that were tailored to fit a specific need share more details on the implementation and how that could benefit that particular scenario.

Constrained OS - 5 out of the 13 solutions support multiple constrained OSs that can be used on the edge devices. The other solutions either do not provide information regarding this matter or share the use of one specific OS. These OSs are typically some kind of real-time operating system. RIOT OS, in particular, is one popular choice used in resource-constrained devices, 2 out of 13 solutions.

Constrained language - Most of the solutions found use a specific programming language to develop the programs run on the constrained devices. However, 4 out of 13 solutions support multiple different languages. Nevertheless, we can also assume that ANIARA Project [30], which uses WebAssembly, implicitly supports multiple languages since, as mentioned previously, WebAssembly binary can conventionally be obtained by converting code from almost any programming language.

Software as a Service (SaaS) - Only 4 out of 13 solutions, only in grey literature, are denominated as Software as a Service since they represent already commercial products. These solutions require a constant internet connection, which can be a limitation in some applications where stable connectivity is an issue due to physical constraints, for example.

Scalability (S) - 4 out of the 13 finds do not show sufficient details to draw any conclusion regarding how their solutions scale in real environments. Those who provide more details regarding this matter do not explicitly mention the concurrent number of devices they support. Regardless, by reading their documents and documentation, it is clear that all of those *probably support many*. The only two solutions that provide this information clearly are

Guerin *et. al* [22] and Silva *et. al* [45] whose testing demonstrate that using 10 devices and 50 devices, respectively, is possible and the system still behaves according to the defined specification and constraints.

Fault-tolerance (FT) - Almost half the solutions found, 5 out of 13, do not share details on how and if they implement fault-tolerance mechanisms. Other solutions, like Tiny-MLOps [3], do fault tolerance by re-adapting the system once a failure occurs; the same type of approach is used in GeneSIS [17] although that re-adaption behaves differently and has a different impact in the system. ANIARA Project [30], on the other hand, provides device redundancy by creating redundant resources that are used in case of failure. Moreover, one solution that more explicitly provides details on how they implement fault-tolerance is Silva *et. al* [45] where they demonstrate how their task reallocation and device redundancy work in their extensive multiple-scenario testing. On the other hand, to implement fault-tolerance features on Azure IoT Hub [5] or AWS IoT Greengrass [25], it is necessary to install additional add-ons and configure the system to recover when some fault occurs.

Telemetry (T) - A little over half the solutions, 8 out of 13, provide tools to extract telemetry and device status data from the network. The commercial products in our grey literature (excluding Toit) are arguably the best in showcasing this data as they often provide graphing agents, alerts, and other tools to track the devices in the network.

3.4 Conclusion

After analyzing all of our results from our systematic literature review and our grey literature, we consider that there are already some very well-documented solutions for the orchestration of applications in resource-constrained devices for the IoT. However, these solutions tend to focus on specific aspects and issues. This can be easily observed as some use a particular programming language or OS on edge development. This is a limitation since it restricts the developer on what he can or cannot use despite streamlining the development process. Surprisingly, having multi-language or/and multi-OS support can also be a disadvantage since it complicates the deployment and forces the developer to create these concrete pipelines for his use case.

However, as mentioned previously, ANIARA Project [30] uses WebAssembly, which allows the usage of an already provided WebAssembly pipeline, which means there is no need to create one ourselves while allowing the developer to create his applications in any programming language since these can be converted back to WebAssembly using existing tools. Still, this solution is incomplete, as they do not share any details on fault-tolerance or scalability. One paper that appropriately shares these details is Silva *et. al* [45] by providing extensive testing and scenario showcase. However, it does not provide any orchestrator redundancy.

Besides, only ANIARA Project [30] out of these two provides some telemetry information. Yet, the telemetry data provided is quite limited. Other solutions like GeneSIS [17] or Golioth [21] are quite more complete as they provide graphing tools and alerts. But, the enterprise solutions

Azure IoT Hub [5] and AWS IoT Greengrass [25] are industry standards and the actual state-of-the-art concerning this matter. Despite that, these enterprise solutions are quite complex and require a set of specialists to set up for each specific application.

With all of these observations in mind, it is possible to conclude there is a gap in the literature in view of the fact that our objective is to produce a complete solution that (1) is straightforward and dynamic to use, hence the idea of using WebAssembly from the start, (2) includes fault-tolerance and scaling mechanisms in the form of orchestrator redundancy, device redundancy, and task reallocation, and (3) provides extensive telemetry data that is easy to setup and use to evaluate the entirety of the IoT ecosystem.

Chapter 4

Problem Statement

4.1 Problem	30
4.2 Desiderata	32
4.3 Hypothesis	33
4.4 Research Questions	34
4.5 Scope	34
4.6 Proposed Solution	35
4.7 Validation and Evaluation	35
4.8 Summary	36

In this chapter, we will be exploring the problem statement for this dissertation work. In Section 4.1, we present the current issues with solutions on the state of the art and previous works. In Section 4.2, we define the desiderata for this project related to the previously mentioned problems. In Section 4.3, we state the hypothesis for this thesis. In Section 4.4, we transform the previously defined hypothesis into research questions. In Section 4.5, we specify the scope of our work, mentioning what we aim and do not aim to do. In Section 4.6, we briefly explain the proposed solution that will be performed. Finally, in Section 4.7, we briefly explain the methodology used to validate and evaluate the engineered solution.

4.1 Problem

Reiterating and extending what was preliminary mentioned in Section 1.2, p. 3, in the IoT, low-power and resource-constrained devices are becoming increasingly popular for their small form factor and efficiency. However, this poses one first problem: this decrease in power at the edge of these networks prohibits these devices from having very complex systems within. Typical Unix-compatible solutions like Docker [15] for containerization, Kubernetes [32] for orchestration of containers, and Argo CD [8] for continuous development are not available in situations like the aforementioned problem as these constrained devices do not even support any type of full-size

operating system. In recent years, though, new solutions have surfaced for these kinds of devices, like Toit [1] and Golioth [21]. Even so, these provide only some of the features we seek in this dissertation.

While exploring the state of the art solutions, we also had the opportunity to more deeply explore some of the works that preceded the one we will be producing. Initially, we noted this in our motivation, Section 1.3, p. 1.3, our dissertation will be advancing on the next steps of innovation as suggested by Pisco *et. al* [37] in their thesis's future work as they thought it would allow us developers to demonstrate the tool's full capabilities better. Besides, their work, Wasmico, already fitted a lot of the prerequisites we had for this work, such as the use of WebAssembly as the task language.

Moreover, there have been some works such as Silva *et. al* [43, 45] where they tackle integrating an orchestration system into a system based on the visual programming Node-RED. This work was then extended by Costa [12], Fragoso [19], and Silva [44] in their dissertations. All of these works focused on different flaws drawn out in [45], mainly when it came to optimal task allocations and different approaches to achieve successful and efficient orchestration of tasks across heterogeneous devices. However, a visual approach like this one was dropped in favor of one that was built from the ground up and did not have the limitations Node-RED and Micropython had. Hence, we are not following the visual programming approach in this work.

Although all of these solutions improved the initial one, they never addressed a technology they always depended on: MicroPython [35], which, despite being faster than regular Python for microcontrollers, is still slow compared to WebAssembly [54], for example. Not only that, but it also created the dependability we aim to remove, as all applications need to be developed in that particular language. Nevertheless, all of these works, notably the latest ones, provide helpful insight into how these complex decentralized task orchestration systems work and how certain mechanisms are implemented for the IoT. Therefore, we will take inspiration from them throughout this dissertation.

The concern regarding solutions like the two mentioned at the start of this chapter, Toit [1] and Golioth [21], is further intensified by our findings in our state of the art in Chapter 3, p. 14. For these two and the other grey literature solutions, the problem typically lies in these tools' usability and orchestration features. Although most of them provide great telemetry tools and are incredibly versatile in the technologies they support, they tend to be complicated to use and do not provide mechanics like fault tolerance and auto-scaling for orchestration. Also, some of them require constant internet connection, which is a limitation in scenarios where accessibility to such service is poor, or non-existent.

Still, the solutions found throughout our systematic literature review still pose a different set of problems that curiously contrast with the ones in the grey literature. This is because most of the solutions found, despite providing details regarding their orchestration features, are typically limited to one specific technology, such as the programming language used to program the constrained devices or the actual device platforms supported while sharing limited telemetry information, if any.

To sum up, this essentially divides the current issues with state of the art solutions in two contrasting aspects:

- Complicated and challenging to set up, provides primitive orchestration features, and requires a constant connection to the internet;
- Limited in terms of functionality and applicability in real-life scenarios, often confined to one particular technology or development pipeline and limited telemetry information.

Interestingly, this means that these limitations contrast with the approaches that exist in orchestration on the cloud as these solutions tend to be so uncomplicated to deploy and provide so many tools to manage the devices and their tasks.

In more detail thought, the problems found in our state of the art altogether are:

1. **Limitation to one programming language** since it limits the developer on developing its application in one particular language that sometimes is not optimal for these resource constrained devices [36, 6, 3, 22, 44, 1, 7];
2. **No extensive orchestration fault-tolerance and/or auto-scaling mechanisms** since these are essential nowadays as devices in IoT keep increasing in demand and their management is therefore increasingly difficult. [2, 36, 6, 3, 22, 30, 21, 1, 7];
3. **Complicated and challenging set up** as tool complexity is growing and new features are introduced unnecessarily, increasing the complexity of the systems [5, 25];
4. **Limited telemetry information and display** is necessary to evaluate and study the system accurately [6, 22, 44, 1, 7];
5. **Constant connection to the world wide web** can be a limitation in some situations where there are physical constraints to such access [21, 1, 5, 25].

At this moment, with our initial decision to base our work on the Wasmico [42] tool, the first issue will be partially tackled since it uses WebAssembly [54], which implicitly allows the use of any programming language to produce the programs for the tasks to be run by the constrained devices.

4.2 Desiderata

Considering the limitations previously mentioned and other related requirements, our solution should fulfill the following desiderata:

D1 Multiple programming language support. This tool’s compatibility with multiple programming languages will be fundamental in achieving any system’s best possible versatility and portability. We aim to provide this feature in a way that does not bother the developer with using more tools, frameworks, or other complicated pipelines, hence the idea of using Was-mico [42] as the basis for our work since it uses WebAssembly [54], as previously mentioned. WebAssembly applications have the advantage of being generated from any programming language using already available tools online.

D2 Over the air updates and auto deployment. As this will allow a system to be automatically deployed and updated over the air (OTA) according to a defined architecture on a configuration file.

D3 Robust task orchestration with fault-tolerance and auto-scaling mechanisms. Although this encompasses many steps, we want to build a system that can orchestrate and distribute tasks across a broad list of devices whose capabilities fit the task they are provided with. With that, we also want to include mechanisms within the system’s orchestration that can reorganize or readapt the system if, for whatever reason, any of these tasks or devices fail, for example.

D4 Telemetry features. To demonstrate the status of each device and tasks running on it. Overall, this will allow us to track and measure important metrics, such as memory and CPU usage on each device.

However, we should note that this last point (D4) is just deemed as desirable as it is not priority compared to the others.

4.3 Hypothesis

In this dissertation, the work that will be revealed in the following chapter aims to validate the following hypothesis:

“In large scale IoT systems, a solution that reliably orchestrates tasks over the air across multiple devices will reduce system deployment times and allow it to automatically adapt in case of failures of any of these devices or the tasks themselves by redeploying and balancing them to meet system demands.”

“In large scale IoT systems” we mean systems that scale either in terms of devices, devices diversity, or features. In “orchestrates tasks over the air across multiple devices”, we mean that the system needs to be able to organize itself and have new tasks/commands deployed over the air automatically across a series of heterogeneous devices. In “reduce system deployment”, we mean this when compared to a manual approach. The last part of our hypothesis, “automatically adapt in case of failures of any of these devices or the tasks themselves by redeploying and balancing them to meet system demands” is already self-explanatory.

So, to sum up, this thesis aims to produce an orchestration system that can organize tasks across a large IoT system. We aim to shorten the deployment times and increase the reliability and autonomy of the system overall by providing mechanisms that reorganize the system in case of any problem, particularly when any of these tasks fail, or any of the devices themselves fail too.

4.4 Research Questions

Based on the hypothesis stated above, we formed the following three research questions. We will use them to validate our work in order to prove the hypothesis above:

RQ1 *Can we provide the ability to orchestrate and dispatch tasks over the over-the-air (OTA) with the capacity and capabilities to run them while saving time and human resources?* To allow the developer to define a high-level architecture of the system and be able to deploy the tasks to just one source, the orchestrator, that can autonomously dispatch the task to a valid device according to its constraints, saving on time and human resources.

RQ2 *Can we increase the reliability of the IoT system by providing a mechanism that can consistently keep the system coherent when a task fails?* To allow the system to continue working even when a task fails by restarting it, the most likely scenario, or redistributing it to another capable device.

RQ3 *Can we increase the reliability of the IoT system by providing a mechanism that can consistently keep the system coherent when a device fails?* To allow the system to continue working even when a device fails by redistributing the tasks across other devices and even discarding some, according to some priorities if necessary.

We further develop, albeit succinctly, how we are testing and answering these questions in Section 4.7.

4.5 Scope

Although the scope of task-oriented orchestration, and especially DevOps works, can be extensive, we decided to limit ourselves to developing a system that can orchestrate devices and tasks with limited fault tolerance and scalability features. This means that the system produced will have constrained device redundancy and will be able to perform task relocation if anything on the system goes wrong, according to some specifications; however, it will not have any orchestrator redundancy.

On the same note, something that is also out of scope for this work is semantic information query, where, for example, if a request cannot be fulfilled directly on any device directly, the system can, if possible, gather information from other devices that cannot perform that same action, to produce an equivalent result.

4.6 Proposed Solution

The solution that will be developed in this dissertation should successfully fulfill the hypothesis mentioned above. It should include all the software components necessary to produce a plausible solution that can be tested and evaluated according to some parameters. Moreover, it is vital to note that, once again, the solution should be based on the current version of Wasmico [42] available on GitHub¹. Our end goal is to create additional modules that allow the orchestration of devices according to their capabilities and computational capacity and integrate these new features into the existing CLI, extending it into the TUI afterward. Refer to Chapter 5, p. 37.

4.7 Validation and Evaluation

In software engineering, there are many ways to conduct research and experiment. Stol *et. al* [47] list guidelines for conducting research in software engineering. Specifically, strategies that should be used throughout these research practices - Field Study, Field Experiment, Experimental Simulation, Laboratory Experiment, Judgment Study, Sample Study, Formal Theory, and Computer Simulation.

When it comes to the validation process, we will be performing a field study research strategy that will directly capture the environment and all the variables the system depends on in a real scenario, such as the physical failure of the devices and other physical constraints that are hard to simulate with emulation, for example.

Notably, we will validate our solution by presenting a series of different scenarios where our system will be responsible for orchestrating a group of tasks across various devices. These experiments will be performed using exclusively physical devices since these provide more realistic results to actual applications of our systems. They will validate whether the orchestration was successful, especially if the system created confines to the given specification. This means we will primarily focus on validating if the system fits the scalability and fault-tolerance requirements and answers the research questions and, inherently, the hypothesis.

Secondly, to evaluate our solution, we will perform tests regarding how long it takes for the system's tasks to be deployed and initially orchestrate and re-orchestrate in case of failure. In order to perform these tests, we will initially be using our physical devices. However, we will also use virtual devices to explore more complex scenarios that cannot be completed with the limited devices at hand. This will provide us with measurements of deployment and (re)orchestration times of what a real-life scenario would look like if we had sufficient devices with us. This will evidently be part of an experimental simulation research strategy. Refer to Chapter 6, p. 53.

¹<https://github.com/SIGNEXT/wasmico-ng>

4.8 Summary

In this chapter, we presented the problem with current solutions and our planned solution, albeit at a very high level and quite briefly.

In the first section, Section 4.1, p. 30, we present the issues with current solutions, particularly those found during our state of the art study. In the meantime, we also explored some of the works that anticipated this one to draw inspiration and evaluate their flaws and how we are tackling them.

In the second section, Section 4.2, p. 32, we list the desiderata for this work, namely, (a) Multiple programming language support, (b) Over the air updates and auto deployment, (c) Robust task orchestration with fault-tolerance and auto-scaling mechanisms and (d) Telemetry features.

In the third section, Section 4.3, p. 33, we present the main hypothesis and in the fourth section, Section 4.4, p. 34, based on this hypothesis we formulate the research questions that we aim to answer fully at the end of our work.

In the fifth section, Section 4.5, p. 34, we share the scope of our work, particularly the topics we will explore and will not explore in the next chapters.

In the sixth section, Section 4.6, p. 35, we briefly share our proposed solutions and that we will base the work on the currently available micro containerization software for low-end embedded systems – Wasmico. More details in Chapter 5, p. 37.

Finally, in the seventh section, Section 4.7, p. 35, we showcase the validation and evaluation methodology that will be used in this dissertation. More details in Chapter 6, p. 53.

Chapter 5

Solution Overview

5.1 Wasmico	37
5.2 Orchestration System	40
5.3 UI Adaptation	50
5.4 Summary	52

Along this chapter, we will be sharing the details about our solution and its implementation over an existing system. In the first section, Section 5.1, we explore the existing system, Wasmico, we are going to be working and iterating on. Then, in the second section, Section 5.2, we comprehensively detailed our orchestration algorithms, basic and Consensus-Based Bundle Algorithm (CBBA). Finally, in Section 5.3, we detail the integration and the changes made to the Wasmico Terminal User Interface (TUI) base system to accommodate these algorithms.

5.1 Wasmico

As previously mentioned, our work will be based on Wasmico [42], a tool that provides the containerization of WebAssembly [54] applications on ESP-32 like devices. We ultimately made this choice to expand on Wasmico based on the fact that it already provided a solid foundation for developing an orchestration system on low-end microcontrollers with over the air updates. Although we have already gone through Wasmico before in Section 2.3, we will further delve into its architecture and features over the following sections to provide a better context to what we will be working on.

As expected, Wasmico’s architecture is straightforward since the devices it can be used on have limited resources and capabilities to support much more than that. It practically consists of two main components: FreeRTOS , a real-time operating system [20], and Serial Peripheral Interface Flash File System (SPIFFS), a filesystem for Serial Peripheral Interface (SPI) NOR flash devices and embedded systems [16].

In FreeRTOS, a scheduler is responsible for ensuring the multiple tasks run over it run concurrently according to some time constraints. By default, the system launches an HTTP web server task that is responsible for initializing an HTTP web server and receiving connections from the host machine via the Command Line Interface (CLI) and cURL, among others. The other tasks running under FreeRTOS are WebAssembly applications that the developer can create and initialize using the HTTP server.

However, these WebAssembly applications do not run natively on the microcontroller, also known as bare metal. Therefore, an interpreter is necessary to create this abstraction layer. In Wasmico, Wasm3 [52] was used for its lightweight footprint and simplicity. So, when a new task is started a new Wasm3 runtime is created as a task with the WebAssembly code uploaded to be run.

Finally, the SPIFFS filesystem is solely responsible for managing the files within the device and enabling it to save the task state and the WebAssembly task files.

Throughout the works of Ribeiro [40], in the first instance, and Pisco [37], in the second instance, Wasmico developed multiple features that will be fundamental in conducting our work. These are the following:

Task Upload [40] Uploads a WASM file into the device's SPIFFS filesystem. This feature also allows the developer to specify an additional four parameters: `reservedStackSize`, `memoryLimit`, `reservedInitialMemory`, and `liveUpdate`. The first one refers to the memory that FreeRTOS should reserve for the later execution of this task, the second and third ones refer to compilation parameters in the compilation of WASM, and the last one is a boolean that indicates if the task should be updated and restarted when a new version of itself is later uploaded.

Edit Task Details [40] Edits one task already uploaded into the devices. It is possible to change the same parameters as the ones defined in its upload.

Task Creation [40] Creates a new task based on the parameters previously defined. Internally, it creates a new instance of the Wasm3 runtime on a new FreeRTOS thread where the WASM functions will be interpreted and finally run by the chip. With Pisco [37], since new state files can be loaded into the devices, when a new task is created, the state can be extracted, if it exists, to start the task from there instead.

Task Stoppage [40] Stops a task already running by stopping the FreeRTOS thread running it, freeing all the resources reserved for its execution.

Task Deletion [40] Deletes a task, its file and all the parameters that were input on its upload.

Task Pause [40] Pauses a task saving all the information regarding its current execution and state so it can restore it when it resumes. After this, the task gets deleted, and all resources are freed except the ones generated when pausing, similar to when a task gets deleted. However,

this behavior was changed when Pisco [37] changed it only to use the FreeRTOS function to suspend the task instead of all the previous process.

Task Unpause [40] or Task Resume [37] Unpauses a task restoring the state and running information at the time of the pause. This process includes the creation of a new FreeRTOS thread to run the task, similar to when a task is created from scratch. Like the pause function, Pisco [37] changed this behavior only to use the FreeRTOS function that resumes the task previously suspended/stopped. They also decided to change the name of this functionality from *Unpause* to *Resume* for coherence with the FreeRTOS function. This is also how we will mention this functionality throughout the rest of this thesis.

List Task Details [40] Lists all the information regarding all the files running on one device. This information includes all the values defined when uploading these files to the microcontroller.

List Active Tasks [40] Lists all the currently running tasks on one device. For each of these tasks, it shows if it is paused or not. If it is not paused, it also shows the lowest free memory stack space the task has ever not used during its execution.

Get Heap Info [40] Shows the user information regarding the microcontroller's current memory status and FreeRTOS. It serves two values: the size of the free heap and the size of the largest continuous block that is free on the heap.

Device Status/Ping [37] Checks if a device is online and currently running Wasmico.

Restarting the Microcontroller [37] Restarts one of the microcontroller devices which terminates all running tasks and frees all the memory allocated to them.

Save the Task's State [37] Saves the state of a task. The state is generated by a task function that returns bytes with the data it intends to save as state.

Upload the Task's State [37] Uploads the state of a task. The state is saved on the target device and will be overwritten if another state upload is done.

All of the features previously mentioned will be important for our work. Particularly, those developed by Pisco [37] in his master thesis will be of utmost importance. Specifically, the implementation of the task state operations and restarting the microcontroller will be fundamental in the scope of features we aim to develop.

Wasmico already has most of the essential and appropriate tools to produce an orchestration system based on it since it first allows all operations to be done over the air, including device restarts, and second because it is, for example, possible to transfer tasks from one device to another without losing the task's current state. This feature can be handy when we want to move a task from one device to an equivalent one to release space for a second task to run on the first device, something we call task reallocation and load balancing.

Last but not least, as we have mentioned multiple times across this dissertation, WebAssembly (WASM) is the basis of Wasmico, making it the perfect candidate to explore, work on, and build upon.

5.2 Orchestration System

As previously mentioned, the first part of our implementation will be an orchestration system. The orchestration system will be responsible for recognizing the devices and distributing tasks across them according to the developer's demand. This distribution should consider multiple factors and variables that will be explored throughout the remainder of this section.

5.2.1 Orchestration Prerequisites

As previously mentioned, the task distribution across the devices will consider multiple factors and variables, which we will refer to as constraints from now on. To build such constraints, Wasmico must provide information regarding the device itself on a low level. Currently, Wasmico only gives details regarding memory usage on the microcontroller and its running processes. However, it does not provide information regarding the external capabilities of each device, Input/Output (I/O) connections and connectivity, for example.

In our case, and for our application and testing, we are only interested in knowing which type of information the device can access through its I/O connections. For example, if the device has a temperature sensor connected, it should have the *capability* `temperature`; if an oxygen sensor is present, it should have `oxygen` instead. The same device can obviously contain multiple *capabilities*.

In Wasmico, we changed the default device status request on the `wasmico` endpoint to return the device capabilities. These capabilities are currently statically defined in the Wasmico runtime when it first executes to simplify our testing and development. Defining these values according to the connected I/O components is out of the scope of this thesis but could be interesting to explore in a later development.

The process diagram of requesting and obtaining the capabilities through the `wasmico` endpoint of a device can be found in Figure 5.1. An example of such request successful response can also be found in listing 5.1.

```
1 {  
2   "status_code": 0,  
3   "data": "capability1, capability2, capability3",  
4   "length": 37  
5 }
```

Listing 5.1: Ping request (wasmico) example response.

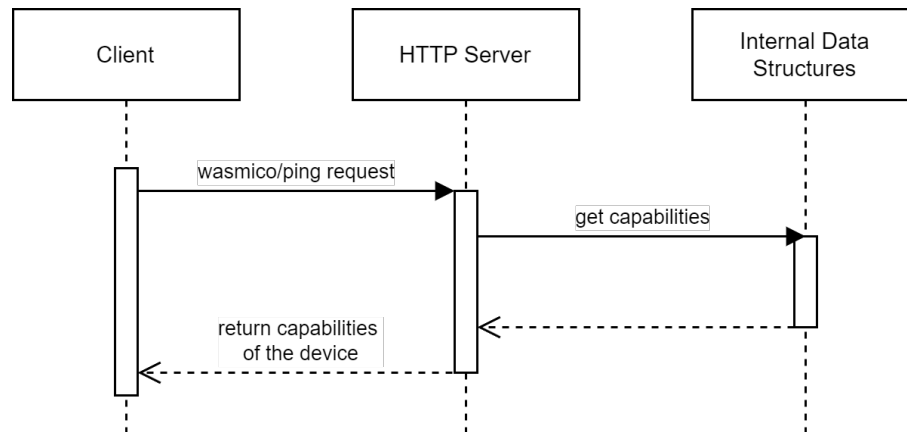


Figure 5.1: Ping request (wasmino endpoint) operation sequence diagram update.

Also, due to how some of the algorithms below will behave, we also decided to include the maximum device memory on the `Get Heap Info` endpoint call, resulting in a response like the following in listing 5.2.

```

1 {
2   "free_heap_size": 203723,
3   "largest_free_block_size": 132921,
4   "max_memory": 262144,
5   "status_code": 0
6 }
  
```

Listing 5.2: Get Heap Info request example response.

5.2.2 Orchestration Declarative Configuration File Specification

In orchestration systems akin to Kubernetes [32], declarative files that define the orchestration system's architecture and behavior are typically used. These files define parameters ranging from the algorithms that will be used to orchestrate tasks to the tasks that will be executed. In each of these tasks, for example, some other parameters might be defined related to their requirements such as IP connections to an Application Programming Interface (API), mounted disk and memory, for example.

In order to define this configuration file, we opted into using a YAML file [55]. This choice was based on other similar popular orchestration solutions that also use it to define the architecture and behaviors of their systems like Docker [15], Docker Compose, and Kubernetes. Not only that, but this approach also serves as documentation for how our system is designed and its purpose. So, like other products, we declaratively define our system on this file, although in a much simpler format without much configuration due to the simplicity of the current version of Wasmico. Listing 5.3 shows an example of what our file looks like.

```
1 task1.wasm:
2   parameters:
3     - 10000
4     - 65536
5     - 4096
6     - False
7   numberOfTasks:
8     - 3
9   constraints:
10    - TemperatureSensor
11 task2.wasm:
12   parameters:
13     - 50000
14     - 65536
15     - 4096
16     - False
17   numberOfTasks:
18     - 1
19   constraints:
20    - TemperatureSensor
21    - OxygenSensor
22    - Wi-fi
23 task3.wasm:
24   parameters:
25     - 5000
26     - 65536
27     - 4096
28     - True
29   numberOfTasks:
30     - 5
31 task4.wasm:
32   parameters:
33     - 20000
34     - 65536
35     - 4096
36     - True
37   numberOfTasks:
38     - 5
39   constraints:
40    - Wi-fi
41   extraScore:
42     - 2
```

Listing 5.3: Orchestration system configuration example yaml file.

As seen in listing 5.3, the structure of the configuration file for the Wasmico orchestration system is relatively simple compared to solutions like the previously mentioned ones. First, we define the filename of the task we want to execute. Within it, we define the parameters that will be passed on task creation, more specifically, the ones described in Section 5.1: `reservedStackSize`, `reservedInitialMemory`, `memoryLimit` and `liveUpdate` in this order. Then, we define how many of these tasks will be run across all devices, considering that one device cannot run the same task simultaneously (Wasmico limitation). Finally, we may or may not define these tasks' operational requirements and an extra score value that will later be exclusively used by one of the algorithms.

The orchestration file can be input on an upgraded version of the currently implemented TUI (Terminal User Interface), which now includes both our orchestration algorithms that will be discussed across two of the following Subsections 5.2.3 and 5.2.4. Each of these algorithms will utilize the previously discussed values differently to build and keep the system coherent with the defined architecture.

5.2.3 Basic Algorithm

The first algorithm is the most naive approach to this task allocation problem. It takes into account three simple constraints and iteratively assigns the tasks to each device with no regard to scalability, load-balancing, or priorities. The decision to first implement such a simple algorithm was based on creating a simple base and proof of concept for the feasibility of building an orchestration system based on Wasmico container running devices.

First, the algorithm collects all the information regarding what files to orchestrate, their parameters, number of tasks, and constraints as provided by the yaml orchestration file described in the previous subsection, Subsection 5.2.2. It does not use the `extraScore` value even if it is defined as this value is related to the calculation of the score function on each task allocation on each device by the Consensus-Based Bundle Algorithm (CBBA) on the following subsection, Subsection 5.2.4. After collecting this information, the algorithm begins a periodic task (every one second, for example) that performs the following steps as shown in the sequence diagram in Figure 5.2.

1. Collects all the currently running tasks in all devices;
2. Calculates, according to the currently running tasks and all the tasks that the system aims to orchestrate, the missing tasks to be run by the devices;
3. Using this list, iteratively go through all devices and every task and verify three basic constraints:
 - (a) Verify if the device is not running this task already;
 - (b) Verify if the device has enough memory to allocate the new task comfortably;
 - (c) Verify if the device has all the features that fit that task's operational requirements.

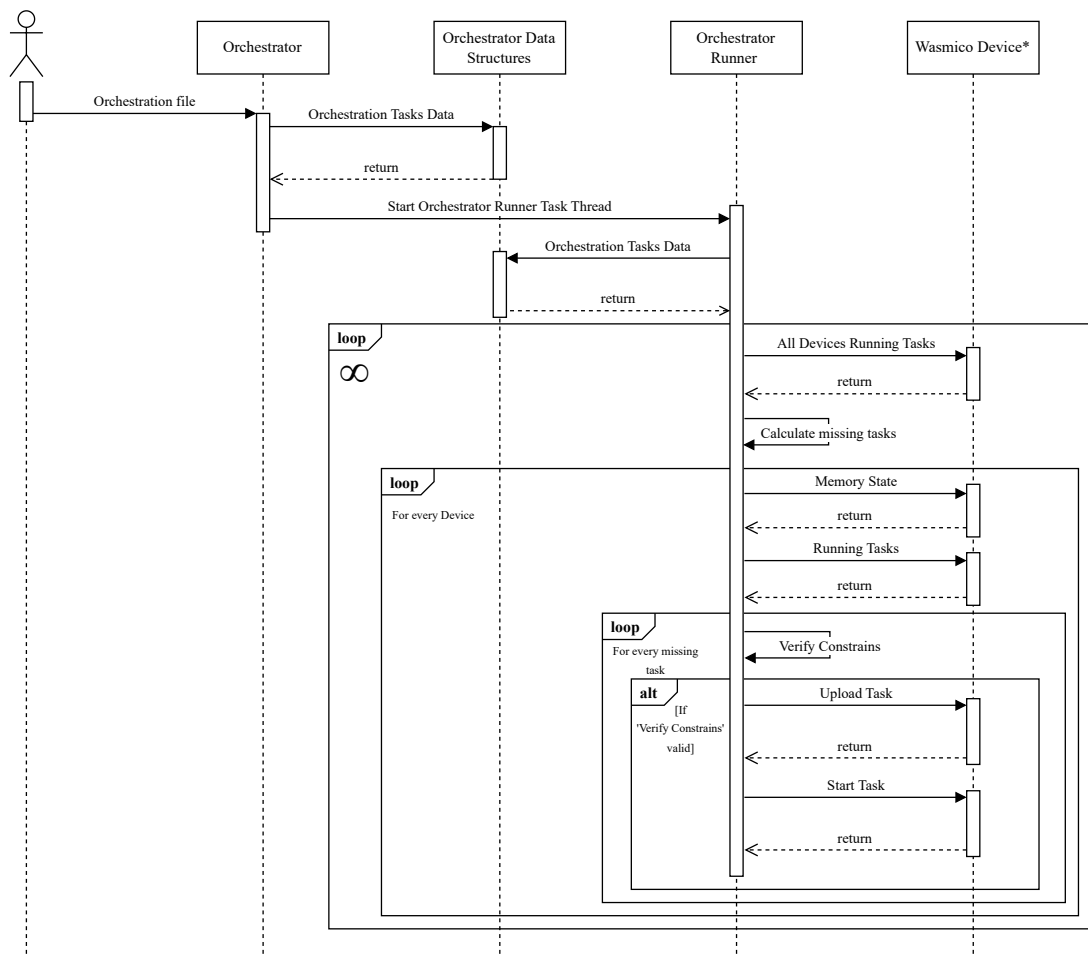


Figure 5.2: Orchestration basic algorithm sequence diagram.

As this process is periodically being executed, whatever happens to the system regarding device or task failures is supervised, allowing it to quickly act, depending on this task's period, to these changes and allocate the missing tasks to other devices if any are available, behaving sort of a 'resource monitor.' This behavior encompasses two types of self-healing mechanism [14] – a device redundancy and runtime adaptation mechanisms where some devices are redundant to the ones that failed, tracked by the previously mentioned 'resource monitor,' to receive the missing tasks to complete the reinstate the previous stable state.

However, due to this algorithm's simplicity and iterative nature, it is not guaranteed that the most number nor the most important of tasks (priority) will be allocated to all devices. Also, whenever a new device is added to the pool, and all the tasks are already running, the system is unable to reallocate and re-balance them across all devices. In the diagram in Figure 5.3, we can observe an example where these limitations are evident.

In this scenario, two identical devices, one with more memory (device 1) and one with less (device 2), are placed inside a closed system where two equally demanding yet different tasks are in play. Only device 1 is turned on in the first state, and both tasks are successfully allocated. However, the system does not balance when we turn on device two in state two. In state three, we

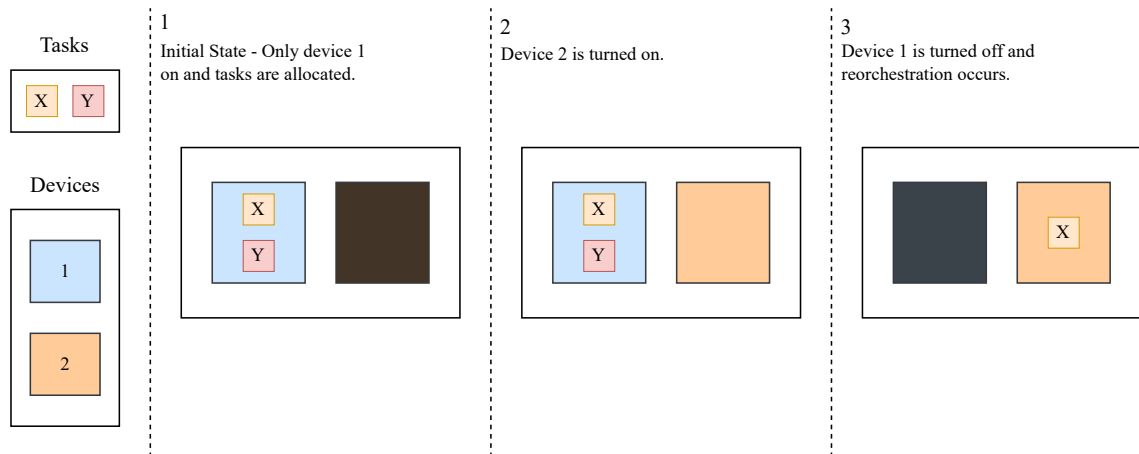


Figure 5.3: Example limitation scenario for the basic orchestration algorithm.

turn off device 1, causing only one of the tasks to be transferred to device 2 due to its memory limitations. However, despite both tasks having similar constraints, there is no guarantee that the most relevant or memory-intensive task was the one allocated to this device, as this attribution is solely based on the order in which the tasks are declared.

5.2.4 Consensus-Based Bundle Algorithm (CBBA)

As per the second and more complex algorithm for self-organizing tasks across multiple devices, the Consensus-Based Bundle Algorithm (CBBA) [10] was chosen. The algorithm uses an iterative approach comprising a bundling and a consensus phase where the devices first choose which task appropriately fits their constraints and bids the highest score according to some scoring function. Then, the winning devices share their beliefs, their winning bids, and their respective scores with others. If there are any conflicts with these received beliefs, these devices that have bids lower than the current new winning bid need to update their beliefs accordingly, which in turn results in them having to remove not only the task from their bundle but also all the following ones while updating other structures.

The choice of this algorithm was made based on the results of previous works, particularly the one proposed by Silva [44] on a Wasmico-like system based on MicroPython [35] and Node-RED [24] by Silva [45]. They proved the algorithm's efficiency, simplicity, and effectiveness in handling multiple tasks and devices simultaneously while performing the allocations in a balanced way, taking into account multiple constraints, which we will mention throughout this subsection and add onto them. However, contrary to them we will not be implementing the ACBBA (Asynchronous Consensus-Based Bundle Algorithm) [31] variant/improvement as it did not present that much of an advantage to being implemented over the base CBBA as, according to preliminary benchmarks, in similar testing scenarios, our implemented version of the CBBA already executes in less than a second, which is considerably lower than what was previously performed with an ACBBA approach on Silva's works [44].

Therefore, along this subsection, we will essentially base our system on the one produced by [44] with some changes based on our architectural needs.

Last but not least, we will also be performing comparisons between the previous systems and ours in Chapter 6.

5.2.4.1 Problem Definition

As we did not find it appropriate to explain the task allocation problem previously in-depth on the last algorithm's implementation due to its simplicity, it is essential to do so now so the CBBA can be better understood across the following subsections.

A task allocation problem basically comprises a list of tasks N_t and devices/agents N_d , the main objective of which is to find the best combination of tasks and devices to maximize some score function previously defined c . Since we can deploy multiple tasks of the same kind to different devices, represented as vector T_j where each index contains the number of instances for task j , we present this mathematical definition different from the traditional form as the following integer optimization problem below in expression 5.1 and constraints in expression 5.2.

$$\max \sum_{i=1}^{N_d} \left(\sum_{j=1}^{N_t} \sum_{k=1}^{T_j} c_{ij}(\chi_i) x_{ijk} \right) \quad (5.1)$$

Where, x_{ijk} represents if task index j instance index k is allocated to device index i . c_{ij} is the score function for allocating task index j on device index i taking into account the bundle that is currently formed for that device χ_i .

Subject to the following constraints.

$$\sum_{k=1}^{T_j} x_{ijk} \leq 1, \quad \forall (i, j) \in \mathcal{I} \times \mathcal{J} \quad (5.2a)$$

$$\sum_{i=1}^{N_d} x_{ijk} \leq 1, \quad \forall j \in \mathcal{J}, \forall k \in \mathcal{K}_j \quad (5.2b)$$

$$x_{ijk} \in \{0, 1\}, \quad \forall (i, j) \in \mathcal{I} \times \mathcal{J}, \forall k \in \mathcal{K}_j \quad (5.2c)$$

Where, aside from the common definitions stated above, the index sets are defined as $\mathcal{I} \triangleq \{1, \dots, N_d\}$, $\mathcal{J} \triangleq \{1, \dots, N_t\}$ and $\mathcal{K}_j \triangleq \{1, \dots, T_j\}$, $\forall j \in \mathcal{J}$. These restriction on 5.2 represent (a) the limitation that each device index i combination with task index j can only run one of its instances at most, (b) the limitation that each task index j and each of its instances k can only run in a total of one devices at most and (c) the limitation that the variable x_{ijk} can only be a binary value (0 or 1) representing if task index j instance index k is allocated to device index i (1) or not (0), as previously mentioned.

5.2.4.2 Algorithm

As previously mentioned at the beginning of this section, the CBBA consists of an iterative process divided into two phases - bundling and consensus. However, our bundling phase adds only the best first task to avoid more conflicts later.

During these two phases of the algorithm, three vectors are manipulated and shared between all devices as the algorithm runs synchronously and therefore, the devices do not need to keep an individual notice of each of these vectors - b_i representing the current bundle on device index i , wd_{jk} representing the winning devices for task index j instance index k and wb_{jk} representing the winning bid for task index j instance k .

With this in mind, this iterative synchronous process works in the following way as shown as pseudocode in Algorithm 1:

Algorithm 1 CBBA

```

procedure CBBA()
   $changes \leftarrow \mathbf{true}$ 
  while  $changes$  is  $\mathbf{true}$  do
     $changes \leftarrow \mathbf{false}$ 
     $I \leftarrow 1$ 
    while  $I \geq N_d$  do
       $J \leftarrow \text{GetBestTask}(I)$ 
      if  $c_{IJ}(J) > 0$  then
         $changed \leftarrow \mathbf{true}$ 
         $b_I \leftarrow b_I \oplus_{end} J$ 
         $K \leftarrow \text{argmin}_k(wb_{Jk})$ 
         $wb_{JK} \leftarrow c_{IJ}(J)$ 
        if  $wd_{JK} \neq -1$  then
          Consensus( $I, J, K$ )
        end if
         $wd_{JK} \leftarrow I$ 
      end if
       $I \leftarrow I + 1$ 
    end while
  end while
end procedure

```

So, firstly, we go through all available devices and gather the best task that currently fits each of them. If a task has a score higher than 0, it is added to that device's bundle. Besides that, the new task is also added to the winning bids and device vectors, considering that this task will replace the instance whose current bid is the lowest. In case that happens, the algorithm knows it should keep running and proceed to the consensus phase, where it assumes there was a disruption in the stability of the matching. This means the algorithm will keep running iteratively until no more tasks can be allocated to any devices.

As mentioned earlier, the bundling portion of our algorithm does not complete a full bundling, but rather, it just adds the first best task it finds before proceeding to the consensus phase. We did

this as it reduced the number of swaps and consensus operations needed to reach a stable system since, this way, the tasks are already naturally distributed across the devices. In pseudocode, this can be represented in the following form in Algorithm 2.

Algorithm 2 GetBestTask

```

procedure GETBESTTASK(i)
  return  $\text{argmax}_j (c_{ij}[b_i \oplus_{\text{end}} j] \times h_{ij})$ 
end procedure
  
```

where

$$h_{ij} = \begin{cases} 1 & \text{if } c_{ij} > \min_k (wd_{jk}) \\ 0 & \text{otherwise} \end{cases} \quad (5.3)$$

Equation 5.3 means that it is only relevant to consider this task index j on device index i as a candidate for its bundle if the score obtained is **strictly greater** than the minimum of bids for all instances across that task.

Finally comes the consensus phase, which only happens if a task is added to a device despite already belonging to another bundle. For example, if we have Device 1 that added a new task with a score of 10, then when it comes to the Device's 2 turns, and it obtains a score of 20 on that same task, it will be necessary to remove that task from Device 1 bundle and all tasks added after that one. While doing so, we must remove those entries from the winning bid and device vectors. This process is necessary to maintain the coherence between all data structures, from bundles to auxiliary vectors, and guarantee convergence by making fewer and fewer changes as more iterations are completed.

5.2.4.3 Score Function

Also, to guarantee this convergence, the original paper [10] mentions that the score function must fulfill a property anointed **Diminishing Marginal Gain** (DMG), which means that the score must decrease as the bundle increases in size. Mathematically, it can be presented in the following way in equation 5.4.

$$c_{ij}[b_i \oplus j] \leq c_{ij}[b_i] \quad (5.4)$$

Regarding the score function itself, our approach follows closely what was previously implemented by Silva in their thesis [44] in regards to its formulation as it still hoists the following properties:

1. Handles predicates and properties, namely constraints regarding capabilities and memory limitations;
2. Handles a balanced allocation of tasks across devices using the **min-first** approach;

3. Handles device instability;
4. Handles device-node instability.

First, the score function evaluates if the device in question has the capabilities to run a certain task. It also verifies if the device currently contains enough memory to hold this new task. Mathematically they can be represented in the following way in equation 5.5.

$$c_{ij} = 0 \quad \text{if} \quad \sum_{bj \in b_i} \text{ReservedStackSize}(bj) + \text{ReservedStackSize}(j) > \text{TotalMemory}(i) \quad (5.5a)$$

$$c_{ij} = 0 \quad \text{if} \quad \text{Constraints}(j) \not\subseteq \text{Capabilities}(i) \quad (5.5b)$$

The decision to just include the **min-first** balanced allocation approach from the previous works was made because, in our case and test scenarios, it is critical to allocate as many tasks as possible, saving as much memory as possible to reduce failure [14]. However, if we want to boost one task's priority, we can do so by providing the `extraScore` field on the orchestration configuration YAML file. We opted to calculate this value in a decimal percentage, between 0.0 and 1.0, to avoid overvaluing devices with more total space. Considering the above, the **min-first** score equation can be defined as the following in 5.6.

$$c_{ij} = (\text{TotalMemory}(i) - \sum_{bj \in b_i} \text{ReservedStackSize}(bj) + \text{ReservedStackSize}(j)) / \text{TotalMemory}(i) + \text{ExtraScore}(j) \quad (5.6)$$

Finally, we also introduced instability variables that modify the score function according to some parameters. For device instability, we introduced a system similar to previous works where we limit the total memory of a device. Hence, the algorithm allocates fewer tasks to it to diminish failure damages and overhaul task downtimes. To do this, we reduce the number of total memory a device has in the calculations by a certain percentage according to the number of failures the device has. Meaning, we modified expression 5.5a in 5.7. Inherently, this allows our algorithm to isolate some devices from the orchestration and circumvent [14] their usage by resorting to others.

$$c_{ij} = 0 \quad \text{if} \quad \sum_{bj \in b_i} \text{ReservedStackSize}(bj) + \text{ReservedStackSize}(j) > \text{TotalMemory}(i) * \text{Stability}(i) \quad (5.7a)$$

where

$$Stability(i) = \begin{cases} 1 & \text{if } Fails(i) = 0 \\ 1 - (1 + e^{-k(Fails(i)-m)})^{-1} & \text{otherwise} \end{cases} \quad (5.7b)$$

Regarding the pairings of device-task instability, we measured the number of times a particular task failed on each device and recorded them. We then performed an operation similar to the one previously shown previously. Yet, instead of reducing the maximum memory capacity of a device by a percentage, we decided to reduce the score directly. Thus, those pairings are less likely to happen, prioritizing others. Therefore, we modified 5.6 as seen in 5.8. Again, this mechanism also allows the system to avoid certain combinations of devices with some tasks to avoid any future failures and downtimes in the future [14].

$$c_{ij} = ((TotalMemory(i) - \sum_{bj \in b_i} ReservedStackSize(bj) + ReservedStackSize(j)) / TotalMemory(i)) * TaskStability(i, j) + ExtraScore(j) \quad (5.8a)$$

where

$$TaskStability(i, j) = \begin{cases} 1 & \text{if } TaskFails(i, j) = 0 \\ 1 - (1 + e^{-k(TaskFails(i, j)-m)})^{-1} & \text{otherwise} \end{cases} \quad (5.8b)$$

In the two previous equations, two variables k and m define the stability function. They define the function curve steepness and midpoint, respectively.

Finally, regarding our particular implementation within the Wasmico ecosystem, it is critical to note that in order to keep the system within the specification provided by the user on the configuration file, the CBBA algorithm is run every second. The devices and their tasks are also monitored during that time frame to allow for reorchestration and task reorganization as fast as possible, diminishing downtimes. This means that, as the basic algorithm, this one also contains a sort of ‘device monitor’ providing redundancy and runtime adaptation mechanisms [14] to be utilized in case of failures.

5.3 UI Adaptation

As previously mentioned, we decided to incorporate our orchestration system into the currently implemented TUI (Terminal User Interface). So, due to all the changes previously mentioned, tweaks were made to some of its core functionalities.

For starters, the previous implementation saved on a plain text file in persistent memory information regarding the devices added to the system. However, in order to maintain data regarding

the number of device fails and task-device fails used on the CBBA orchestration algorithm, we changed the format to a JSON file, as shown in the example in listing 5.4.

```

1  [
2      {
3          "name": "testpython0",
4          "ip": "0.0.0.0:8080",
5          "group": "default",
6          "fails": 1,
7          "taskFails": {
8              "task2.wasm": 5,
9              "task3.wasm": 1,
10             "task5.wasm": 7
11         }
12     },
13     {
14         "name": "testpython1",
15         "ip": "0.0.0.0:8081",
16         "group": "default",
17         "fails": 2,
18         "taskFails": {
19             "task1.wasm": 2,
20             "task2.wasm": 1,
21             "task3.wasm": 2,
22             "task4.wasm": 3
23         }
24     }
25 ]

```

Listing 5.4: Terminal User Interface (TUI) devices JSON file.

The options for the two algorithms were also added to the initial prompt, where the user can choose which file they wish to use for the orchestration. Moreover, if the orchestrator ever fails and is later restarted by the user, a mechanism was implemented enabling the system to rebuild its previous internal state and resume the orchestration process [14]. Finally, due to the introduction of device capabilities as shown in Subsection 5.2.1 as well as the device and device-task fails previously described, it only made sense to always keep these up to date. So, the process of pinging the device, retrieving the currently running tasks, and updating these data structures are run on every iteration of either of our algorithms, so about every second, as previously mentioned.

Finally, due to the implementation of an emulated version of Wasmico, there was a need to integrate the IP port into the logic related to the device connection, allowing multiple emulation instances to be run on the same device simultaneously.

5.4 Summary

The implementation for what was showcased across this chapter can be found on a public GitHub repository¹.

In the first part, Section 5.1, p. 37, we reviewed some details of the base technology used, Wasmico, what it offers, and its relevance for the rest of our work.

In the second part, Section 5.2, p. 40, details about our orchestration system and its inner workings are revealed. However, first, some changes to the Wasmico framework were made to provide data needed on the orchestration system itself. Then, the declarative orchestration file specification was demonstrated with an example. Finally, we provided thorough details, including mathematical and theoretical ones, of both our orchestration algorithms, basic and CBBA.

More specifically, in this part, thorough details regarding the solution developed were presented, which we codenamed Wasmikube. Wasmikube came out as a solution that was based on a previous approach based on multitask orchestration of devices in IoT by Silva [44] with Node-RED and Micropython. However, as we delved deeper into Wasmico and this solution, we made some changes to their approach and learned some lessons, such as changing the approach to be utterly synchronous on the orchestrator and not overloading the devices with the task of auctioning and consensus, as we believed our devices would not support such a load together with the Wasmico framework. Besides, doing this reduced the auctioning and consensus times as little slow wireless communication between the devices and the orchestrator is needed. Finally, we also chose just one of their converging score algorithm, which, in this case, tries to balance the tasks across devices, allocating the biggest task in the set first and then the rest. We did this because we assumed bigger tasks were more important.

Finally, in a shorter Section 5.3, p. 50, we gave details on the changes made to the current existing TUI to fit both our orchestration algorithms.

¹Github Repository: <https://github.com/SIGNEXT/wasmikube>

Chapter 6

Evaluation and Validation

6.1	Wasmico Python Wrapper	53
6.2	Experiments	55
6.3	Discussion	65
6.4	Summary	67

This chapter overviews the experiments that were done to validate our proposed hypothesis. However, initially, in Section 6.1, we go over a new version of Wasmico in Python that can be used on any device with Python installed. In Section 6.2, experiments were performed across a series of scenarios to thoroughly evaluate the system’s functionality and effectively orchestrate a group of tasks across multiple devices while accounting for multiple constraints and limitations. Finally, in Section 6.3, we made some observations regarding the overall results of our solution and presented some threats to the validity of these results.

6.1 Wasmico Python Wrapper

In order to emulate and simulate Wasmico running devices without relying directly on physical devices, a similar Python version of the Wasmico containerization environment was developed. We also developed this version of Wasmico because we currently do not hold enough physical devices for the tests we will perform in the next Section, Section 6.2. This version can run on any system with Python installed and `pywasm3` [53], `aiohttp` and `psutils` as external dependencies.

6.1.1 Overview

At a high level, this Wasmico python wrapper, which we code-named `Wasmico.py`, fundamentally performs the exact same functionality as the original Wasmico for ESP32 devices. Nonetheless, some key differences exist due to fundamentally being based on entirely different programming

languages. Python [39] is a high-level programming language where tools for controlling the operating system (OS), hardware, and particularly memory are scarce. This means that Wasmico.py has some critical distinctions in its architecture to fit these constraints:

- To build the web server, we used the asynchronous package `aiohttp` to build a simple web server that receives connections and executes the requests asynchronously.
- In order to have tasks concurrency, as there is no FreeRTOS option in Python, it was decided that using the built-in package `multiprocessing` was the most appropriate as it allowed us to terminate and pause tasks at will using `psutils`, something that the package `threading` by itself did not;
- We directly use the `os` built-in package to manage files and record tasks;
- As previously mentioned, we use `pywasm3` [53] as an external package, to run the WASM [54] tasks.
- As we are emulating a physical Wasmico device, when we execute the wrapper we need to define a series of device specific details as parameters:

IP Emulated device IP. If ran in the same host machine typically is `0.0.0.0`;

Port Emulated device IP port. Multiple devices on the same host machine can run different ports;

Capabilities Emulated device physical capabilities to be evaluated by the orchestrator;

Max Memory Emulated device max memory size to allocate tasks.

We should note that the device only uses the **Max Memory** value to calculate the current device memory as it subtracts from it the value found on the `reservedStackSize` of all currently running tasks as this value defines the reserved memory space for that task. On an actual physical device, this current memory value would fluctuate due to other changes on the system, such as printing into the console and outputting values into other devices. Regardless, we will not consider this a limitation as the end goal of having a limiting memory factor as a constraint still stands.

As stated above, one of the most significant advantages Wasmico.py brings is that we can deploy multiple heterogeneous devices on the same machine on different network ports. It also allows us to test and evaluate complex scenarios more rapidly over a larger batch of devices without relying on the physical ones and their complex physical setup.

This version of Wasmico in Python can also be found on the same public GitHub repository with the orchestration algorithm, albeit on the `wrapper/` folder¹.

¹<https://github.com/SIGNEXT/wasmikube/tree/master/wrapper>

6.1.2 Limitations

As any other piece of software, Wasmico.py also has its set of limitation. Actually, its set of limitation is an addition to the already existing limitations of the original Wasmico firmware. Yet, even through all of them Wasmico.py still stands as a viable solution to simulate heterogeneous Wasmico-like devices.

Firstly, the Wasm3 library that we are using, `pywasm3`, is outdated since its last update three years ago, resulting in slower runtimes and fewer functionalities. Also, this library lacks documentation, and while it does provide some examples, they do not showcase all functionalities, making it harder to develop and work on.

Secondly, and still slightly connected to the lack of documentation referred to in the previous limitation, functions defined within the WASM files that contain strings (`char*` in C) as parameters or returns do not work as Python does not have control over memory like C or C++ does making it impossible to pass that information down to the runtimes. We did some research on how we could make this work and even explored the source code, but all our tries were unsuccessful without changing the original source, something that was out of our scope for this dissertation. In turn, this means that all the features related to state saving, restoring on-device tasks, and printing strings to the console do not work as they all depend on this limitation particularity.

An argument could be made that maybe another library should have been chosen for this implementation; however, using this specific library seemed the most straightforward, linear, and comprehensive way to create a Python implementation of Wasmico based on the original Wasmico tool source code.

Finally, we did not emulate any Input/Output (I/O) device or external connection except for the web server clients. This would not matter for our experiments, so we did not explore this option as it was also out of scope. Essentially, we wanted a functional copy of Wasmico that could be run on any device and how many instances to perform more complex tests.

6.2 Experiments

In this section, some experiments will be performed to showcase, validate, and evaluate the orchestration solution for Wasmico that was previously presented. However, we will focus our experiments and results on the CBBA-based system as the basic algorithm is not compelling to present here, and, besides, we already presented an example showcasing its straightforward functionality and flaws earlier in 5.2.3. Still, we will present comparisons between both approaches by making slight remarks on the theoretical results the basic algorithm would give us in that particular scenario, as the result of this algorithm is easy to predict and assess.

6.2.1 Device Setup

As for the setup, in the primary basic validation tests, we will use the three physical ESP-32 devices we have installed with the Wasmico firmware. We will resort to our Python Wasmico

variation for the following evaluation test, Wasmico.py, as it will make it easier to make precise measurements and evaluate our system's performance and orchestration capabilities, particularly its ability to scale at will and handle failures.

When it comes to the device that will be handling the orchestration and the Wasmico.py emulated devices it will be a Dell Precision 5540 laptop with an Intel i7-9850H CPU with 32GB of RAM and an NVIDIA Quadro T2000 graphics card running Ubuntu 20.04 as the OS.

6.2.2 Measurements

For the measurements, we opted into using a generous subset of the metrics used in the thesis by Silva [44], which are the following:

Uptime In seconds, the time a device has been online and connected to the orchestration system;

Free Heap Memory The amount of heap memory, in kilobytes (KBs), each device has available to load more tasks;

Number of tasks Number of tasks allocated to each device;

List of tasks List of tasks allocated to each device.

This data is fetched from the orchestrator's perception of each device that registers these previous parameters every second on a Comma-separated values (CSV) file, which we can later load and produce graphs and other relevant analyses.

6.2.3 Scenarios

In Table 6.1, we briefly present the validation and evaluation scenarios we will explore across the following subsections. In this table, we give details regarding our goals with that experiment, a quick summary of what it is about, and the type of devices used, physical or simulated. Some of these scenarios presented in this table are replicas of the ones presented by Silva in their thesis [44] as we will be comparing how both systems perform in similar scenarios when adequate. Moreover, all of these scenarios orchestration yaml files, as per our specification on Subsection 5.2.2, can be viewed over on the same GitHub page as previous times but on the `tui/orchestrationTestScenarios` folder².

6.2.4 Scenario 1 (S.01)

Scenario 1 (S.01) aims to validate and evaluate the system's behavior, particularly in balancing tasks using physical devices, behaving as a simple sanity orchestration test. In total, we have all of our 3 devices and 6 tasks. As all of these are similar to each other, 2 tasks are assigned to each device in order to produce a balanced allocation. Figure 6.1 shows the measured metrics across the whole life cycle of starting the orchestration system and the devices being deployed where we can observe this happening.

²<https://github.com/SIGNEXT/wasmikube/tree/master/tui/orchestrationTestScenarios>

Scenario	Goals	Summary	Type of Devices
S.01	Basic task balancing.	In a system with 3 devices, we will deploy 6 similar tasks.	Physical
S.02	Task balancing when the constraints are complex.	In a system with 3 devices, we will deploy 10 different tasks with complex memory and capability restrictions, multiple instances and priorities.	Physical
S.03	Basic fault-tolerance capabilities.	In a system with 2 devices, with only one on, we will deploy 6 similar tasks and then turn on and off the devices in an alternating form.	Physical
S.04	Basic scalability capabilities.	In a system with 3 devices, with only one on, we will deploy 6 similar tasks and then turn on the remaining devices one by one.	Physical
S.05	Task balancing taking into account a device's limited memory capacity.	Reproduction of the scenario presented in 5.2.3.	Simulated
S.06	Task balancing taking into account one of the device's limited memory capacity, when some task has higher priority, <code>extraScore</code> .	Reproduction of the scenario presented in 5.2.3. However, one of the tasks has a higher priority.	Simulated
S.07	Task balancing and fault-tolerance across devices when some device keep failing repeatedly.	In a system with 4 devices and 36 similar tasks, we will repeatedly fail one of the devices.	Simulated
S.08	Task balancing and fault-tolerance across devices when some device-task pair keeps failing repeatedly.	In a system with 4 devices and 3 similar tasks, we make one task repeatedly fail on one particular device.	Simulated
S.09	Scalability and load-balancing capabilities with a greater number of tasks, their instances, and devices.	In a system with 10 devices, but only one one, and 50 similar tasks with multiple instances, we will turn on the devices one at a time.	Simulated

Table 6.1: Validation and evaluation test scenarios.

The main goal of any of these experiments is to validate and evaluate the system's capacity to perform any more specific goals for each scenario.

As seen in the diagram above, all 6 tasks are rapidly, under a second, distributed across the physical microcontrollers as expected. As these devices are all similar in terms of memory constraints and the tasks are identical, 2 tasks are allocated to each device. In the memory graph, we observe that at device startup, some memory is already being used by the Wasmico framework and web server. As we deploy the tasks, we see this memory increase by about 30KB and keep fluctuating around those new values, increasing slightly over time.

6.2.5 Scenario 2 (S.02)

Scenario 2 (S.02) aims to validate and evaluate the system's behavior in balancing tasks when the constraints are complex. In this case, we have the same 3 devices as last time; however, we have 10 completely different tasks with different constraints. These constraints vary from the memory

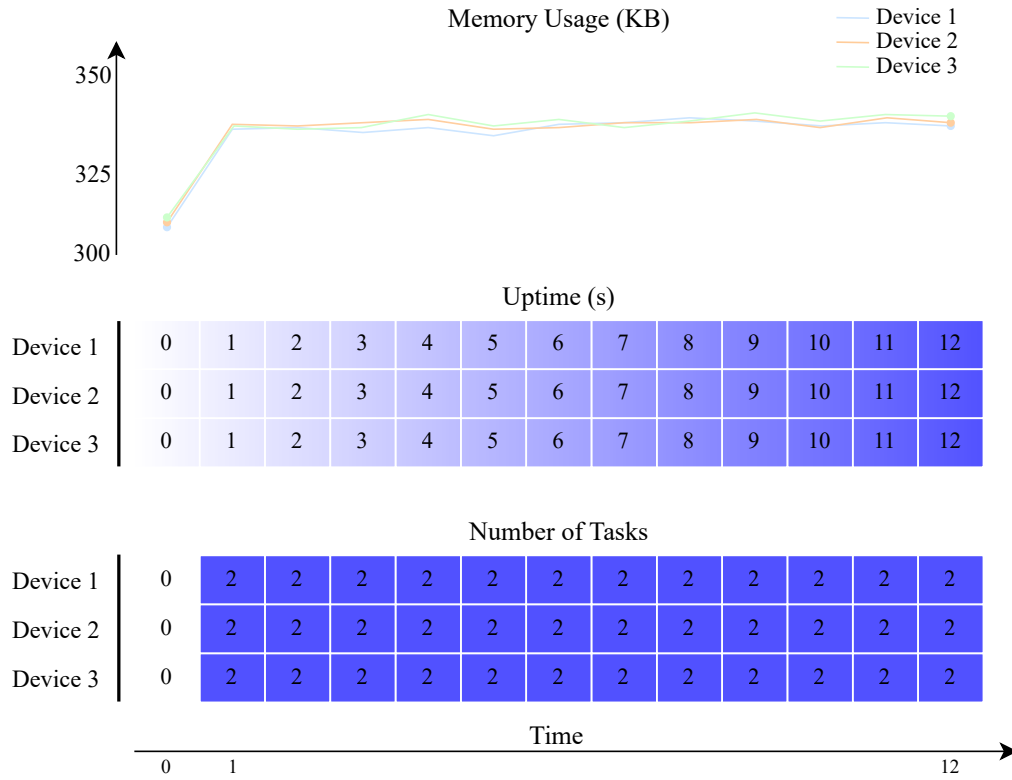


Figure 6.1: Metrics measured during Scenario 1 (S.01) experiment.

required and the number of instances to the physical constraints concerning the sensors the device has on board.

In this particular scenario, we have 2 external sensors to the devices. Device 1 has access to both of them, Device 2 to one of them and device 3 to the other one. In Figure 6.2 we can observe how the allocation of these tasks was made across all devices in the system considering all of these limitations.

We can analyze by this figure that a total of 14 out of the 16 (including instances) tasks were allocated across the 3 devices. Each device has 5, 4, and 5 tasks allocated to it, respectively, after the allocation is completed. However, 2 tasks are not allocated to any devices. Task 2 has 2 instances, one of which is allocated to Device 3. The other instance is not allocated since that task requires a device that has both the temperature and oxygen sensor and as we only have one device with that configuration, one of the instances is not allocated. A similar problem happens with task 8, where we have 3 instances that require an oxygen sensor device, and we only have 2 devices with that capability, leaving one instance out.

It is also important to note that the algorithm produced a balanced and correct task assignment according to all the devices' constraints and memory limitations. The devices have a workload of 36.9, 24.9, and 35.7 kilobytes, respectively, across them, which, according to all the constraints, is mathematically the ideal allocation in this scenario.

Also, in Figure 6.3, we observe what was previously mentioned regarding the allocation of

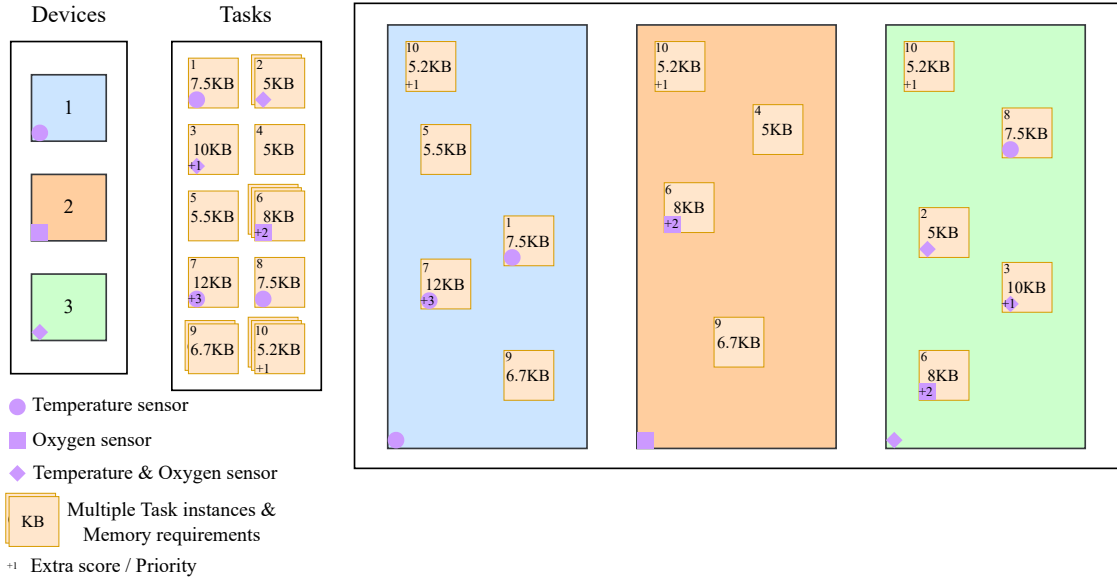


Figure 6.2: Scenario 2 (S.02) experiment system diagram.

tasks across all 3 devices. Even in the memory graph, it is possible to verify that Device 2, which has only 4 tasks allocated, uses less memory than the other 2 devices that allocate 5.

6.2.6 Scenario 3 (S.03)

Scenario 3 (S.03) aims to validate and evaluate the system's fault-tolerance capabilities within the real physical devices realm. In this case, only 2 of the total 3 physical devices are being used, but still the same group of 6 similar tasks as in Scenario S.01, Subsection 6.2.4. Nevertheless, in this scenario, we will evaluate how the system handles the tasks when the only device dies, and another is started sometime after. First, with Device 1, we deploy the system. Then, we turn off this device and turn on Device number 2. We repeat this process, albeit with the device roles reversed. We can observe in Figure 6.4 that when the system has no running devices, no tasks are allocated, but when we start Device 2 the first time, the tasks are all allocated to it. The same behavior happens when we perform the second part of the experiment.

As can be viewed in these graphs, initially, with only Device 2 online, all 6 tasks are allocated to this one device. When Device 1 initiates 2 seconds later, the tasks are distributed across the devices, making each one of them carry a total of 3 tasks. Then, Device 2 turns off, reallocating its tasks to Device 1. Finally, Device 1 turns off, and Device 2 turns back on, making the system return to its initial state with only Device 2 online. However, it is interesting to note that when both devices have 3 tasks allocated to each of them, Device 2 still uses more memory than Device 1 despite the same workload. We predict this happens because although we pause and remove all the tasks from the devices when we reorchestrate, the device still keeps some allocations due to a possible undetected memory leak.

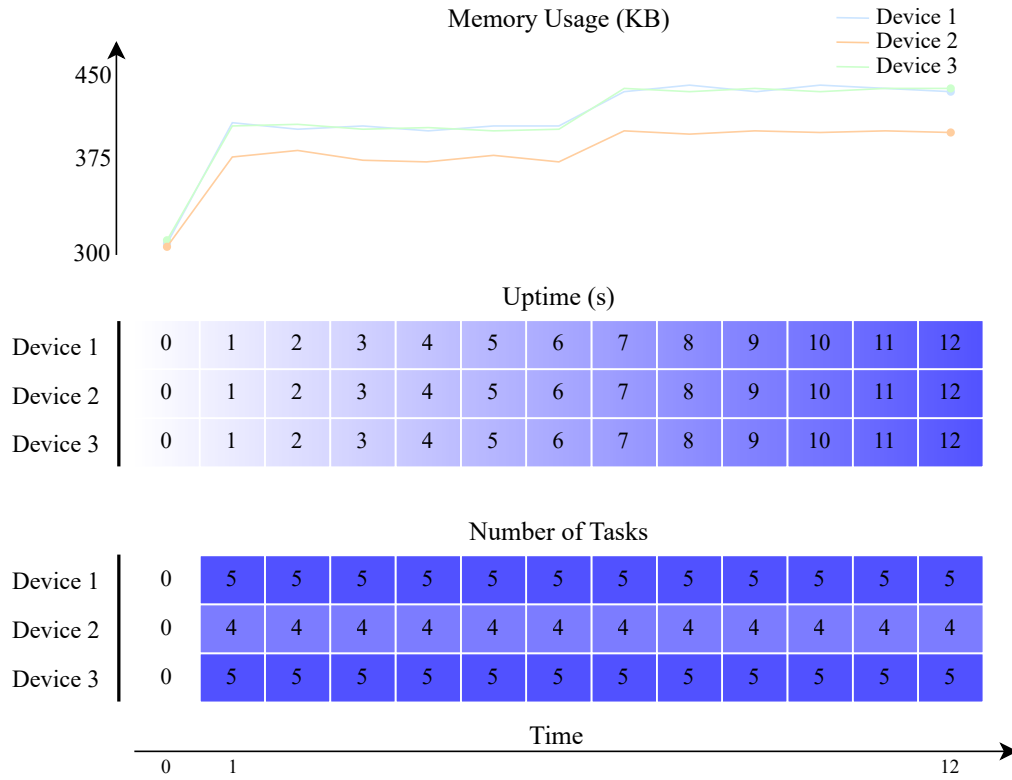


Figure 6.3: Metrics measured during Scenario 2 (S.02) experiment.

6.2.7 Scenario 4 (S.04)

Scenario 4 (S.04) aims to validate and evaluate the system’s scalability capabilities within the real physical devices realm. We have the 3 devices in this case. Again, we will use the same group of 6 similar tasks as in Scenarios S.01 and S.03, Subsections 6.2.4 and 6.2.6. However, in this scenario, we will start by having one device on and then initiate the other devices one by one. We can observe in Figure 6.5 that tasks are balanced out between the devices to produce a balanced allocation as we turn the other devices on.

In this scenario, we begin by having Device 1 on and all 6 tasks allocated to it. As we turn on Devices 2 and 3 at 1 and 6 seconds, respectively, we see the tasks being redistributed across the devices in a balanced matter. Again, as in the previous scenario, we observe the same memory behavior, as although the devices have the exact same workload, devices that previously had more tasks keep a higher memory footprint due to an undetected memory leak in the system.

6.2.8 Scenario 5 (S.05)

Scenario 5 (S.05) aims to validate and evaluate the system’s behavior, particularly in balancing the task load across devices, taking into account a device’s limited memory capacity. We reproduce the scenario presented in 5.2.3 for the basic algorithm. We can observe that contrary to the previous experiment using the later algorithm, the system could balance itself by distributing the

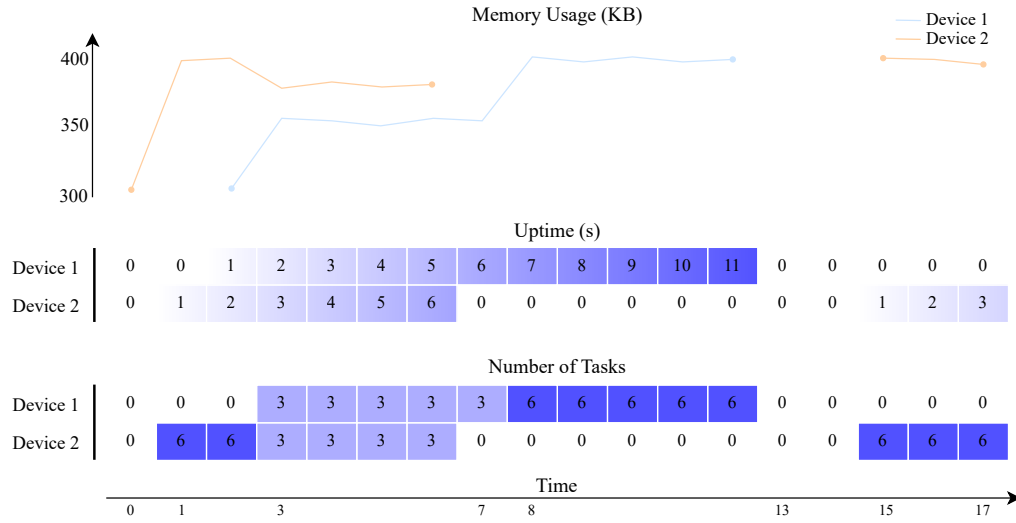


Figure 6.4: Metrics measured during Scenario 3 (S.03) experiment.

two tasks across the two devices when Device 2 turns on. However, the task chosen to stay on the second device was done in the order they were defined in the orchestration file since they are both equally demanding and have no priority. A scenario where we introduce priority and see a change regarding this matter is presented next in Section 6.2.9. In Figure 6.6, we can see a diagram representing the system states as situations emerge, the devices are turned on/off, and the tasks are stopped/crashed, among others.

As stated above, contrary to the findings using the basic algorithm in Subsection 5.2.3, the algorithm is able to reorganize the tasks when a device is turned on or off to keep a balanced allocation across them.

Also, in Figure 6.7 shows the measured metrics across the whole life cycle of starting the orchestration system and the devices being deployed.

As it can be observed, as we turn on the devices, within a seconds delay the tasks are deployed to its respective devices according to the bundle algorithm's result and memory is used as both tasks occupy about 4KB in memory.

6.2.9 Scenario 6 (S.06)

Scenario 6 (S.06) aims to validate and evaluate the system's behavior, particularly in balancing the task load across devices, taking into account the device's limited memory capacity when some task has higher priority, `extraScore`. We reproduce the scenario presented in 5.2.3 for the basic algorithm. We can observe that contrary to the previous experiment (S.05) in Section 6.2.8, the system could balance itself by distributing the two tasks across the two devices when Device 2 turns on. However, the task chosen to stay on the second device is based on the `extraScore` value as it gives priority to a particular task, in this case, task γ^3 . In Figure 6.8, we see a diagram

³Task γ was chosen to be given the `extraScore` because in our previous experiment task x was the task allocated in the last system state and in this scenario we want to demonstrate what happens if γ is priority.

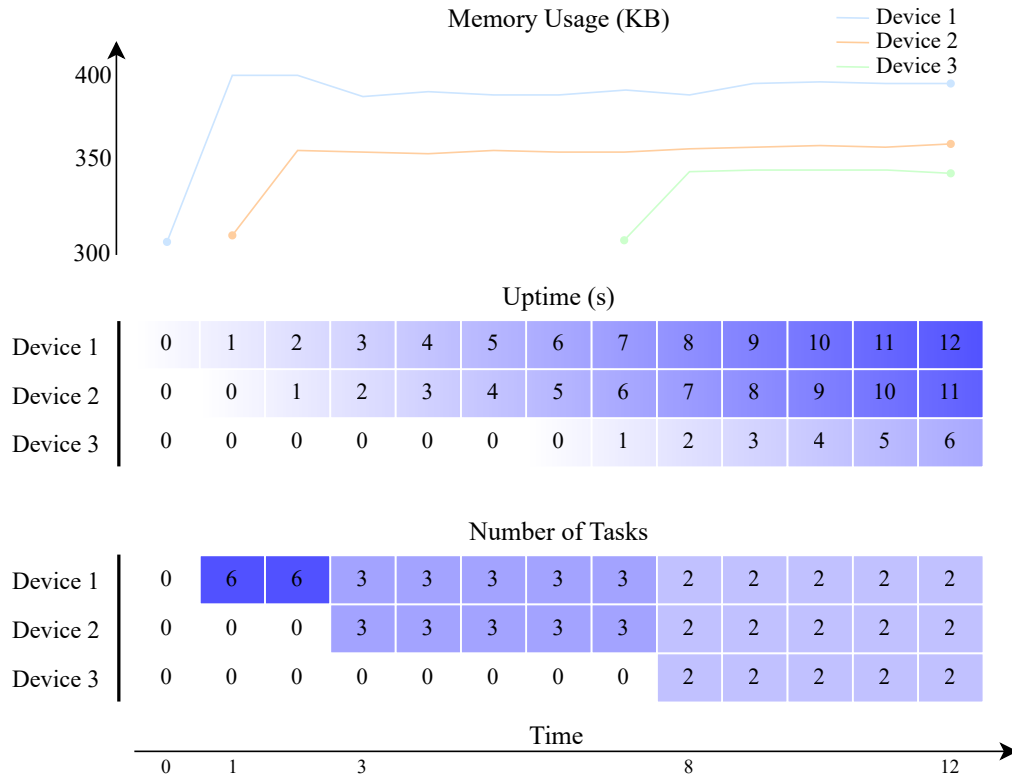


Figure 6.5: Metrics measured during Scenario 4 (S.04) experiment.

representing the system states as situations emerge, the devices are turned on/off, and tasks are stopped/crashed, among others.

As stated above, contrary to the findings in the previous experiment, since task γ now has priority over task x , it is allocated to the second device when Device 1 fails. An interesting side effect of this priority feature is that on Stage 2, instead of transferring task γ to Device 2 like the previous experiment, x is transferred so as not to break γ 's running state.

Figure 6.9 shows the measured metrics across the whole life cycle of starting the orchestration system and the devices being deployed.

In this case, the results are very similar compared to the previous experiment as both experiments just vary in the priority setting of one task and not the tasks themselves.

6.2.10 Scenario 7 (S.07)

Scenario 7 (S.07) aims to validate and evaluate the system's behavior, particularly in balancing the task load and fault-tolerance across devices when some device keep failing repeatedly. Here, we have 4 lookalike emulated devices, and we will deploy the second biggest group of tasks across these experiments, which is 36 similarly demanding tasks. In this experiment, the devices will all start turning on, and all tasks will be allocated in balance to them. Then, we will repeatedly fail and restart one of these devices, in this case device 4, until no more tasks are allocated to it, as we

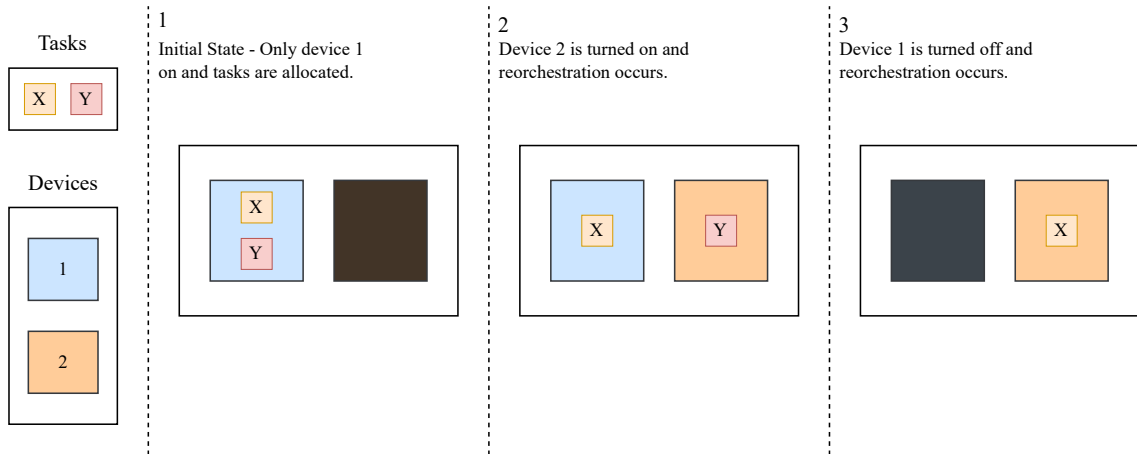


Figure 6.6: Scenario 5 (S.05) experiment system diagram.

expect our algorithm to attribute near 0 scores to devices that have a significant amount of failures, in this case, 6 or more. This exact behavior can be verified in Figure 6.10.

As can be seen in these graphs, we load into our system 36 tasks. Once the devices are first started, allocations are performed within a second, which is an improvement compared to a previous system made by Silva [44] using NodeRED and Micropython as their approach took around 150 seconds to perform the same allocation with a similar group of tasks. After that, as we start failing Device 4 initially after 3 fails, the device still allows all tasks to be allocated to it like normal. Only after that, it starts diminishing the number of tasks it allocates, lowering it to 7 and then 1 after the 4th and 5th fail, respectively. Finally, only after the 6th failure and restart at 28 seconds can the system no longer allocate tasks to device 4.

6.2.11 Scenario 8 (S.08)

Scenario 8 (S.08) aims to validate and evaluate the system's behavior, particularly in balancing the task load and fault-tolerance across devices when some device-task pair keeps failing repeatedly. This scenario plots the same 4 devices as last time but just 3 similarly demanding tasks. We will artificially fail one of these tasks on a device to verify that, even after the first failure, our algorithm will avoid that allocation and try to make it to another device. The graphs and statistics in Figure 6.11 show this exact behavior.

These metrics simply describe what was previously discussed. As it can be observed, when we kill the task on Device 3 at 9 seconds, the orchestrator recalculates the allocations and sends this task to Device 4 instead 2 seconds later at 11 seconds. This essentially just tags these tasks as incompatible with Device 3, rewarding it with an inferior score to what would be expected if no failures ever occurred.

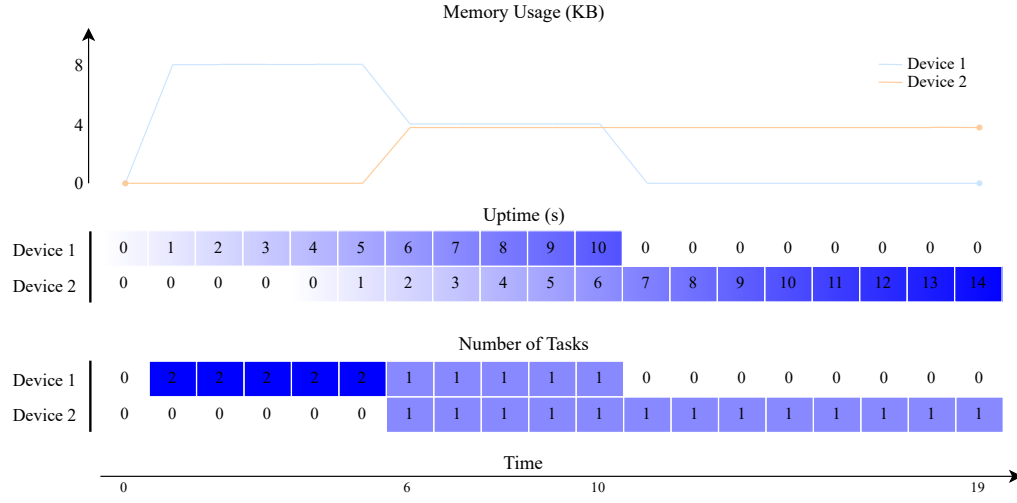


Figure 6.7: Metrics measured during Scenario 5 (S.05) experiment.

The memory usage measurements are merely demonstrative as they do not represent physical devices values.

6.2.12 Scenario 9 (S.09)

Scenario 9 (S.09) aims to validate and evaluate the system's scalability and load-balancing capabilities with a more significant number of tasks, their instances, and devices. Finally, in this experiment, we used the same set of 4 similar devices to deploy the most extensive set of tasks across it, 50 similar tasks with multiple instances so we can have the same task running on 2 different devices. In this case, we will be testing how these are reallocated as we turn on the devices one by one. Firstly, all tasks are allocated to one device, and as we introduce the other, these are distributed accordingly to fit the balanced allocation it is looking for. We can better visualize this happening in Figure 6.12.

It is important to note that although there are 50 similar tasks, there are multiple instances of some of them, making the number of effective tasks higher at a total of 71. The distribution of multiple instances is 1 instance of 38 tasks, 2 instances of 6 tasks, 3 instances of 4 tasks, 4 instances of 1 task and 5 instances of 1 task.

Once again, when a new device is started, the algorithm redistributes tasks around the different devices in about a second or two after initiation, proving again its effectiveness and quickness in such reorganization compared to previous approaches, namely by Silva with Node-RED and MicroPython [44]. We can also observe that instances are also distributed as new devices emerge. For example, when the second device is initiated, 62 tasks are initiated. This is because, besides all 50 tasks as before, the system also starts another 12 from the other groups of instances where there is more than one instance for that task and therefore $(6 + 4 + 1 + 1)$, which is 12. The exact process happens when another device is initiated where 6 more tasks are added $(4 + 1 + 1)$. When the last device is initiated, another 2 tasks are added $(1 + 1)$. As can be noticed, the system is only able to initiate 70 tasks. This happens since, for the 5 instances tasks, only 4 can be created as we only have 4 devices, and no devices can be running the same tasks concurrently, making this task only initiate 4 instances.

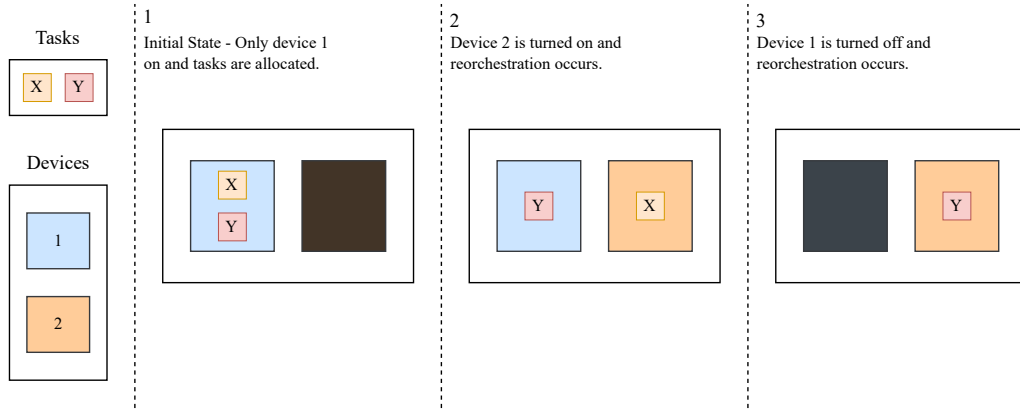


Figure 6.8: Scenario 6 (S.06) experiment system diagram.

6.3 Discussion

In this section, we discuss the main observations taken from all the experiments as well as some threats to validity that presented along the way, which impacted our results and comparisons.

6.3.1 Main observations

Overall, the experiments we performed went over a broad series of scenarios that explored the multiple features of our orchestration system. These experiments went from simple tests to validate if the system was able to do simple orchestration to more complex scenarios. Other experiments also tested the system's ability for our system to automatically scale when new devices or tasks are added to the system. Finally, some of them also tested how the system behaves when a device or task fails and when reorchestration/reorganization of tasks/devices is needed.

In the first four scenarios, we used the physical devices to mainly validate our solution, three similar ones to be exact. We previously concluded that these orchestrations were successful individually with these devices and gave predictable results. Still, something we did not predict would happen was the increase in idle memory footprint as we deployed more and more tasks onto one device. Also, throughout our testing, something that generally happened with this kind of physical device was that we had to repeat the experiments multiple times due to the inherent instability of the Wasmico wireless connection, as referred to as a limitation by Pisco [37] in their dissertation work.

The rest of the tests involving the virtual devices, to perform a more thorough evaluation of our system, were more complex, including up to 4 devices and using their customizability to induce constraints otherwise impossible with physical devices, creating a more controlled testing environment for some specific purposes such as creating a device with limited memory capacity.

With both these types of tests, we were able to conclude that our solution works as initially intended as it allocates the tasks in a balanced manner across the devices, considering their limitations. Not only that but any changes that were made to the system, whether device or task failures, new allocations were made. Also, when a device or device-task pair fails too many times, the

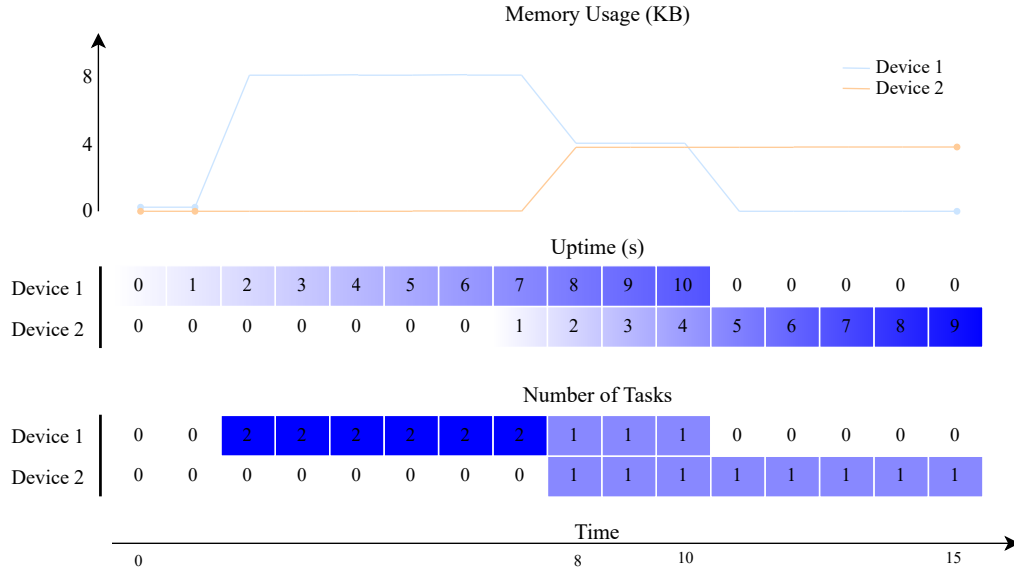


Figure 6.9: Metrics measured during Scenario 6 (S.06) experiment.

The memory usage measurements are merely demonstrative as they do not represent physical devices values.

system is able to revert tasks and allocations to other devices in order to preserve the system’s designed functionality.

As our solution was inspired and some of the experiments were similar to the ones done by Silva’s in their thesis, one of the main objectives for us following such a route was to be able to compare their results more smoothly. It is still important to note that they used an asynchronous version of the algorithm that, despite being centralized on a central orchestrator, the orchestration algorithm was run on each device asynchronously, and a consensus needed to be reached between all of them for the allocations to be completed. Still, ours and their solutions both achieved the same final allocation results as mathematically expected. However, our solution was consistently faster than theirs while auctioning new allocations due to not having the device and communication overhead their solutions had since the devices these tasks perform on have very limited resources both in computing power and memory, meaning we made allocations in most 2 or 3 seconds while theirs took upwards of 100 to 150 seconds [44].

All in all, our solution filled all the requirements we had previously presented, showing that it can clearly orchestrate a list of tasks across a series of different devices running the Wasmico firmware while taking into account different constraints and types of failures. Not only that but its performance is well above another solutions we compared to.

6.3.2 Threats to validity

As previously seen, we used simulated devices to perform some of the tests we intended to practice. They pose the limitation of not accurately representing physical devices despite having the exact same functionality as Wasmico in a physical device. These virtual devices cannot accurately record memory changes when a task is deployed, as seen by the previous memory graphs between

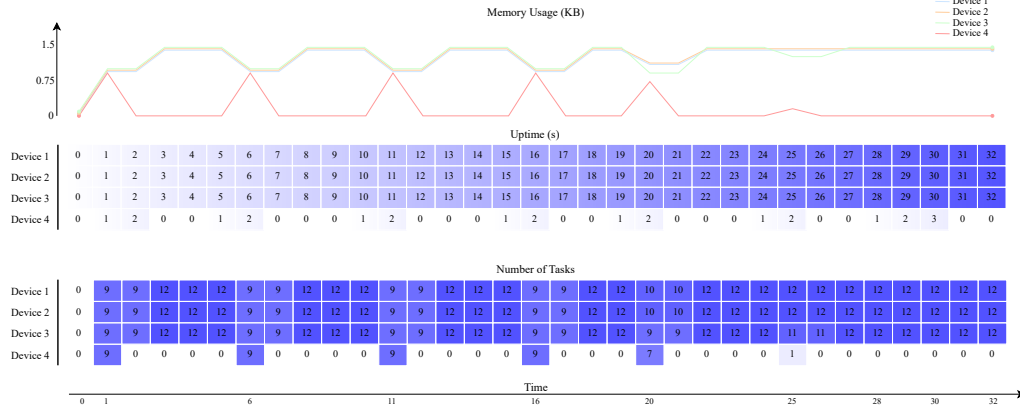


Figure 6.10: Metrics measured during Scenario 7 (S.07) experiment.

The memory usage measurements are merely demonstrative as they do not represent physical devices values.

them. Not only that, but they cannot showcase the network and wireless constraints the physical devices have. Generally, we believe that although these tests are very interesting for understanding the limitations of our work, they do not represent at all a real-world IoT scenario with physical devices.

Compared to previous solutions like the one described in the last subsection, ours does not have as many measurements as theirs, such as current flash memory. Also, as previously stated by Ribeiro in their dissertation, the memory measured by Wasmico is heavily limited by the platform on which it is deployed, sometimes even being mere estimates [40]. However, the other solution in question could display this metric more accurately on their devices.

The limited number of physical devices also posed a challenge in measuring the scalability of our system, as we could mostly just do this with a simulated environment that does not precisely represent reality despite our efforts. If we had more devices, we could test more realistic scenarios with real sensors and tasks that performed real actions. This was something we decided not to pursue due to some limitations on Wasmico, such as its wireless network connectivity instability, which also made us perform some of these tests many times, as previously mentioned, to have all the tasks deployed without error. The more tasks we had, the more bound to happen. These errors were soft limiting the number of tasks we could once again deploy to a physical device for scalability testing. All of this means that if we had more physical microcontrollers with a more stable version of Wasmico and other hardware, we could better test the scalability of our system without relying on virtual devices that do not represent reality the best.

6.4 Summary

Throughout this chapter, we performed a series of tests on physical and virtual devices to validate our orchestration solution for ESP-32 devices.

In the first section, Section 6.1, p. 53, we presented a Python version of Wasmico. This version can be run on any device with Python installed and allows us to rapidly test and tune Wasmico

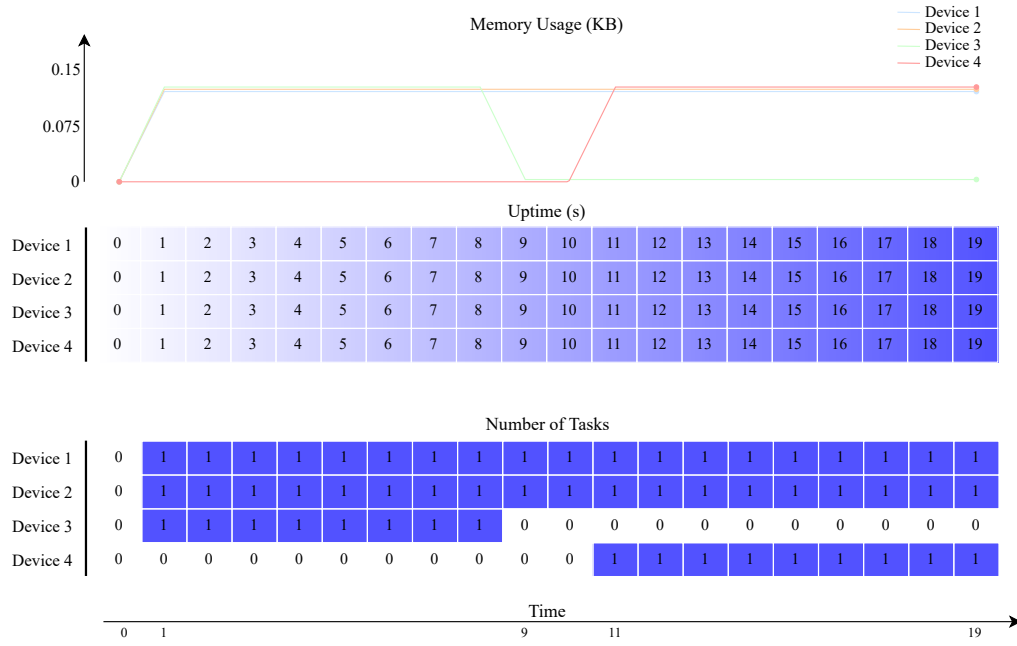


Figure 6.11: Metrics measured during Scenario 8 (S.08) experiment.

The memory usage measurements are merely demonstrative as they do not represent physical devices values.

with much more control over the device's memory and physical constraints.

In the next section, Section 6.2, p. 55, we perform the actual experiments and draw some preliminary, isolated conclusions about the results obtained. However, in general, we would say our results are positive and reflect the effectiveness of our system in orchestrating WASM tasks across multiple devices, physical and virtual.

Finally, in Section 6.3, p. 65, we do a series of general observations regarding all the tests performed, making remarks regarding how it fares against a solution we had our work based on mainly on the algorithm choice and scenarios. We also showcased a series of threats to the validity of our experiments.

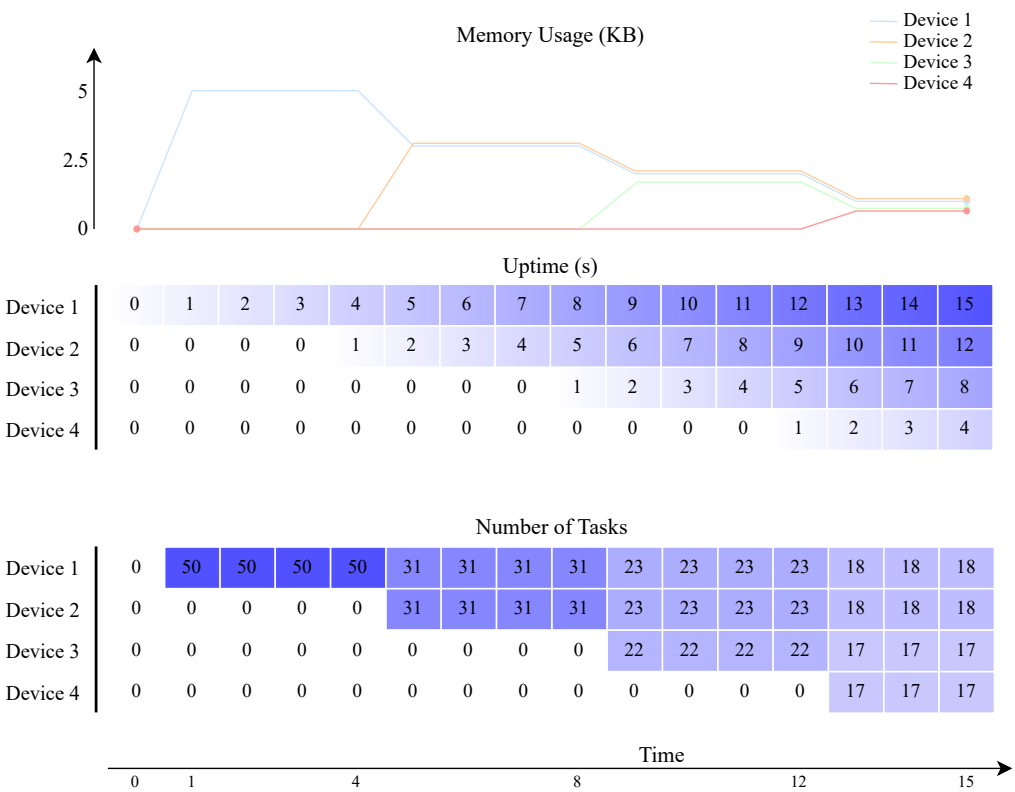


Figure 6.12: Metrics measured during Scenario 9 (S.09) experiment.
The memory usage measurements are merely demonstrative as they do not represent physical devices values.

Chapter 7

Conclusions

7.1	Conclusions	70
7.2	Research Questions Revisited	71
7.3	Future Work	72

In this chapter, the general conclusions of our work are presented. In the first section, Section 7.1, we wrap up what was discussed in the previous chapter, comparing all the tests as a whole while also going over how it behaves compared to other similar systems. In the second section, Section 7.2, we revisit the research questions and hypothesis defined for our work. Finally, in Section 7.3, we go over some work that can help this one improve its current functionality and effectiveness.

7.1 Conclusions

With the increasing number of edge devices and edge-centric networks in the field of Internet-of-Things, challenges have been raised in regard to the management and organization of these devices, which, most of the time, are heterogeneous, low in computational capacity and memory. This heterogeneity makes the deployment of tasks particularly hard. Some devices have different firmware and programming language support as well as libraries, making this process complex and complicated.

Over in Chapter 3, p. 14, we performed a systematic literature review, where we explored orchestration solutions for low-end devices, especially in applications that involved IoT applications. Our findings concluded that along with the grey literature finding, the tools found were heterogeneous in their abilities, meaning that no one could do everything we were seeking, creating the gap in the literature we sought to explore in this dissertation.

This is where our solution comes in. Based on previous developments in the field of containerization for low-end microcontrollers, particularly Wasmico [42] for its use of WebAssembly, a new system was designed in Chapter 5, p. 37, responsible for orchestrating, organizing, and monitoring tasks across a variety of devices. The system aims to take into account the physical

constraints of the devices and changes in their status, such as device failings and task failings. The system is, therefore, responsible for performing the complete deployments and the adjustments necessary to the devices to maintain the system's intended functionality over-the-air (OTA).

The orchestration software built on top of Wasmico uses the Consensus-Based Bundle Algorithm (CBBA) as Silva's solution did in their dissertation [44] using Node-RED and Micropython to produce the task allocations to each device after reaching an optimal solution according to all of our constraints. With the final software developed, multiple tests were performed, as seen in Chapter 6, p. 53, according to a series of different scenarios that explored different parts of our solutions. These experimentations either used physical or virtual devices, with the latter being developed for the purpose of our more extensive testing with more tasks and devices that the physicals did not allow.

Across a series of scenarios we tested every corner of our orchestration system, from simple orchestration to ones with many constraints. We also tested the system behaviour in case of any failures of the devices and tasks. Our solution also outperformed another which we based our orchestration algorithm and some scenarios on, particularly in deployment and allocation generation times.

7.2 Research Questions Revisited

Going back the hypothesis we previously stated in Section 4.3, p. 33:

“In large scale IoT systems, a solution that reliably orchestrates tasks over the air across multiple devices will reduce system deployment times and allow it to automatically adapt in case of failures of any of these devices or the tasks themselves by redeploying and balancing them to meet system demands.”

We produced a set of three research questions that we aimed to respond to after our implementation and testing to prove said hypothesis, whose answers we can give now since we performed our validation and evaluation on the previous chapter.

RQ1 *Can we provide the ability to orchestrate and dispatch tasks over-the-air (OTA) to devices with the capacity and capabilities to run them while saving time and human resources?* Generally, all of our testing scenarios prove that our system can reliably orchestrate and organize tasks across a series of devices with limited capabilities and computational capacity. Not only that, but the allocation is rapidly performed, and the tasks are quickly dispatched to their corresponding devices. Also, as it is built on top of Wasmico, we also clear the ability for this orchestration and task deployment to be done over-the-air (OTA).

RQ2 *Can we increase the reliability of the IoT system by providing a mechanism that can consistently keep the system coherent when a task fails?* Mainly, Scenario S.08 goes over this issue where a task-device pair is failed, and the system is able to continue functioning with no interruptions. Besides, as we fail this task on this specific device, the orchestration system is able to interpret that and allocate that task elsewhere where such failure is less likely to happen according to its own perceptions of the device's previous behavior with that particular task.

RQ3 *Can we increase the reliability of the IoT system by providing a mechanism that can consistently keep the system coherent when a device fails?* A lot of the scenarios we previously presented explicitly have Scenarios S.03, S.05, S.06 and S.07, cover this issue explicitly. In Scenario S.03, for example, we cover how tasks can *ping-pong* from one device to another when they fail in an alternating form. Scenario S.07, on the other hand, shows how tasks are distributed across a series of devices when one fails, not to mention the ability of our system to recognize that device's instability when it receptively fails to allocate fewer and fewer tasks over time until none are given to it.

Thus, given that we could answer precisely all of the questions on which our hypothesis was based, we conclude the developed system can reliably orchestrate tasks over the air across a group of heterogeneous resource-limited devices while automatically adapting its status by moving and stopping tasks across devices in order to maintain balance across the system and meet system demands meaning our evidence supports the hypothesis presented.

7.3 Future Work

Despite our advancements and promising results in developing this orchestration system on top of Wasmico, we found some limitations along the way that need some addressing in future works. Besides, there is also a group of improvements that would increase the reliability and robustness of this system.

State Migration on reorchestration The current version of our orchestration algorithm does not transport the current task state to another device if a reorchestration requires that task to be moved to other devices. As Wasmico already provides tools to save task states and transfer those to other devices, this could be implemented in our orchestrator without changing any fundamental Wasmico logic.

Improve Wasmico connectivity Currently, Wasmico has some connectivity issues that do not allow a reliable and continuous connection to the orchestrator. Some changes in the logic of how this connection is performed and handled can be made to allow these improvements.

Task diversity Where a task would have a backup plan group of tasks that would be initiated if the primary task fails a certain amount of times, this task would need to be a correct substitute for the initial task [14].

System and device checkpoint Devices and the overall orchestration system could use checkpoints to record system state and roll back to previous states if they presented better results in terms of reliability or functionality, for example, [14].

Improve telemetry Although we made some improvements to the telemetry data Wasmico outputted, this was not the focus of our work. However, some improvements in this regard could create further improvements to the orchestration system as most allocation are based on this data.

Auto deployment agent Add some kind of CD agent capable of tracking changes to the orchestration files, and when committed, changes are automatically deployed. Following an approach similar to ArgoCD [8], which we previously mentioned.

Orchestrator redundancy This work was out of scope. However, it would be interesting to implement this as failures at the orchestrator level can happen, and the system needs to maintain functionality and keep its particular configuration state.

Orchestration in devices this size with such tight constraints is complex, and limitations such as their computational and memory capacity need to be taken into account to create a usable and future-proof system reliably. There are several other details that we did not address in this section but in others before this one that can be explored in the future.

References

- [1] Toit . Toit - high-level software platform for the ESP32. <https://toit.io/>. [Online; Accessed 2023-11-26].
- [2] Zabasta Anatolijs, Kunicina Nadezhda, Kondratjevs Kaspars, Patlins Antons, and Caiko Jelena. Development of Sensor System for Legacy and Smart Municipal Systems Infrastructure Control. In *2018 20th European Conference on Power Electronics and Applications (EPE'18 ECCE Europe)*, pages P.1–P.9, September 2018.
- [3] Mattia Antonini, Miguel Pincheira, Massimo Vecchio, and Fabio Antonelli. Tiny-MLOps: a framework for orchestrating ML applications at the far edge of IoT systems. In *2022 IEEE CONFERENCE ON EVOLVING AND ADAPTIVE INTELLIGENT SYSTEMS (IEEE EAIS 2022)*, 2022. Type: Proceedings Paper.
- [4] Arduino. Arduino. <https://www.arduino.cc/>. [Online; Accessed 2023-11-22].
- [5] Microsoft Azure. IoT Hub. <https://azure.microsoft.com/en-us/products/iot-hub>. [Online; Accessed 2023-11-27].
- [6] Emmanuel Baccelli, Joerg Doerr, Ons Jallouli, Shinji Kikuchi, Andreas Morgenstern, Francisco Acosta Padilla, Kaspar Schleiser, and Ian Thomas. Reprogramming Low-end IoT Devices from the Cloud. In *2018 3rd Cloudification of the Internet of Things (CIoT)*, pages 1–6, July 2018.
- [7] Nabu Casa. ESPHome. <https://esphome.io/index.html>. [Online; Accessed 2024-01-02].
- [8] Argo CD. Argo CD. <https://argoproj.github.io/cd/>. [Online; Accessed 2023-10-29].
- [9] Argo CD. Argo CD - Docs. <https://argo-cd.readthedocs.io/en/stable/#architecture>. [Online; Accessed 2023-12-30].
- [10] Han-Lim Choi, L. Brunet, and J.P. How. Consensus-based Decentralized Auctions for Robust Task Allocation. *IEEE Transactions on Robotics*, 25(4):912–926, August 2009.
- [11] Google Cloud. What Is container orchestration. <https://cloud.google.com/discover/what-is-container-orchestration>. [Online; Accessed 2023-11-12].
- [12] Pedro Miguel Sousa da Costa. Decentralized Real-time IoT Orchestration. Master’s thesis, University of Porto, Portugal, July 2021.
- [13] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Designing and constructing internet-of-Things systems: An overview of the ecosystem. *Internet of Things*, 19:100529, August 2022.

- [14] João Pedro Dias, Tiago Boldt Sousa, André Restivo, and Hugo Sereno Ferreira. A Pattern-Language for Self-Healing Internet-of-Things Systems. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–17, Virtual Event Germany, July 2020. ACM.
- [15] Docker. Docker: Accelerated Container Application Development. <https://www.docker.com/>. [Online; Accessed 2023-11-11].
- [16] Espressif Systems. SPIFFS Filesystem - ESP32 - ESP-IDF Programming Guide latest documentation. <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/spiffs.html>. [Online; Accessed 2024-01-28].
- [17] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre-Emmanuel Novac, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. GeneSIS: Continuous Orchestration and Deployment of Smart IoT Systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 870–875, July 2019. ISSN: 0730-3157.
- [18] Fortinet. What Is Fault Tolerance? Creating a Fault-tolerant System. <https://www.fortinet.com/resources/cyberglossary/fault-tolerance>. [Online; Accessed 2023-11-13].
- [19] Tiago José Viana Fragoso. Dynamic Real-time IoT Orchestration. Master’s thesis, University of Porto, Portugal, July 2021.
- [20] FreeRTOS. FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. <https://www.freertos.org/index.html>. [Online; Accessed 2023-11-19].
- [21] Goliath. Goliath instant IoT cloud. <https://goliath.io/>. [Online; Accessed 2023-11-25].
- [22] Fiona Guerin, Teemu Kärkkäinen, and Jörg Ott. Towards a Programmable World: Lua-Based Dynamic Local Orchestration of Networked Microcontrollers. In *Proceedings of the 14th Workshop on Challenged Networks, CHANTS’19*, pages 13–18, New York, NY, USA, 2019. Association for Computing Machinery. event-place: Los Cabos, Mexico.
- [23] Hopsworks. RonDB - World’s fastest Key-Value Store. <https://www.rondb.com/>. [Online; Accessed 2023-12-06].
- [24] IBM. Node-RED. <https://nodered.org/>. [Online; Accessed 2023-12-06].
- [25] Amazon Web Services Inc. Iot Edge, Open Source Edge - AWS IoT Greengrass - AWS. <https://aws.amazon.com/greengrass/>. [Online; Accessed 2023-11-27].
- [26] Amazon Web Services Inc. IoT ESP32 Workshop. <https://catalog.us-east-1.prod.workshops.aws/workshops/5b127b2f-f879-48b9-9dd0-35aff98c7bbc/en-US/module4>. [Online; Accessed 2023-11-27].
- [27] Amazon Web Services Inc. What is Containerization? - Containerization Explained - AWS. <https://aws.amazon.com/what-is/containerization/>. [Online; Accessed 2023-11-11].

- [28] Amazon Web Services Inc. What is Virtualization? - Cloud Computing Virtualization Explained - AWS. <https://aws.amazon.com/what-is/virtualization/>. [Online; Accessed 2023-11-11].
- [29] Jerryscript. Jerryscript: A JavaScript engine for Internet of Things. <https://jerryscript.net/>. [Online; Accessed 2023-11-22].
- [30] Wolfgang John, Ali Balador, Jalil Taghia, Andreas Johnsson, Johan Sjöberg, Ian Marsh, Jonas Gustafsson, Federico Tonini, Paolo Monti, Pontus Sköldström, and Jim Dowling. ANIARA Project - Automation of Network Edge Infrastructure and Applications with Artificial Intelligence. *ACM SIGAda Ada Letters*, 42(2):92–95, April 2023.
- [31] Luke Johnson, Sameera Ponda, Han-Lim Choi, and Jonathan How. Asynchronous Decentralized Task Allocation for Dynamic Environments. *MIT web domain*, March 2011.
- [32] Kubernetes. Kubernetes: Production-Grade Container Orchestration. <https://kubernetes.io/>. [Online; Accessed 2023-10-29].
- [33] Lua. The Programming Language Lua. <https://www.lua.org/>. [Online; Accessed 2023-12-06].
- [34] Somayya Madakam, R. Ramaswamy, and Siddharth Tripathi. Internet of Things (IoT): A Literature Review. *Journal of Computer and Communications*, 03(05):164–173, January 2015.
- [35] MicroPython. MicroPython - Python for microcontrollers. <http://micropython.org/>. [Online; Accessed 2023-11-22].
- [36] Cristina Paniagua, Jens Eliasson, and Jerker Delsing. Efficient Device-to-Device Service Invocation Using Arrowhead Orchestration. *IEEE Internet of Things Journal*, 7(1):429–439, January 2020.
- [37] Xavier Ruivo Pisco. Adopting Containers in Microcontrollers for the IoT. Master’s thesis, University of Porto, Portugal, July 2023.
- [38] Zephyr Project. The Zephyr Project – A proven RTOS ecosystem, by developers, for developers. <https://zephyrproject.org/>. [Online; Accessed 2023-11-25].
- [39] Python Software Foundation. Welcome to Python.org. <https://www.python.org/>. [Online; Accessed 2024-04-09].
- [40] Eduardo Carreira Ribeiro. Micro-containerization in Microcontrollers for the IoT. Master’s thesis, University of Porto, Portugal, July 2022.
- [41] RIOT. RIOT - The friendly Operating System for the Internet of Things. <https://www.riot-os.org/>. [Online; Accessed 2023-12-06].
- [42] SIGNEXT. Wasmico. <https://github.com/SIGNEXT/wasmico-ng>. [Online; Accessed 2023-11-09].
- [43] Ana Margarida Oliveira Pinheiro Silva. Orchestration for Automatic Decentralization in Visually-defined IoT. Master’s thesis, University of Porto, July 2020.
- [44] David Luís Dias da Silva. Adopting Auction-Based Task Allocation Towards Decentralized Orchestration in Mist IoT. Master’s thesis, University of Porto, Portugal, July 2022.

- [45] Margarida Silva, João Dias, André Restivo, and Hugo Ferreira. Visually-Defined Real-Time Orchestration of IoT Systems. In *MobiQuitous 2020 - 17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MobiQuitous '20, pages 225–235, New York, NY, USA, 2021. Association for Computing Machinery. event-place: Darmstadt, Germany.
- [46] Statista. IoT connected devices worldwide 2019-2030. <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/>. [Online; Accessed 2023-11-05].
- [47] Klaas-Jan Stol and Brian Fitzgerald. Guidelines for Conducting Software Engineering Research. In *Guidelines for Conducting Software Engineering Research*, pages 27–62. ResearchGate, August 2020.
- [48] Espressif Systems. Get Started - ESP32 - – ESP-IDF Programming Guide latest documentation. <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/>. [Online; Accessed 2023-11-22].
- [49] TelluIoT. ThingML. <https://github.com/TelluIoT/ThingML>. [Online; Accessed 2023-12-07].
- [50] Toit. Toit programming language. <https://toitlang.org/>. [Online; Accessed 2023-11-26].
- [51] Exploding Topics. Number of IoT Devices (2023-2030). <https://explodingtopics.com/blog/number-of-iot-devices>. [Online; Accessed 2023-11-05].
- [52] Wasm3. Wasm3. <https://github.com/wasm3/wasm3>. [Online; Accessed 2023-11-19].
- [53] Wasm3. Pywasm3. <https://github.com/wasm3/pywasm3>. [Online; Accessed 2024-04-07].
- [54] WebAssembly. WebAssembly. <https://webassembly.org/>. [Online; Accessed 2023-10-29].
- [55] YAML. The Official YAML Web Site. <https://yaml.org/>. [Online; Accessed 2024-03-04].