Università degli Studi di Padova

Department of Mathematics
"Tullio Levi-Civita"

Master's degree in Computer Science

Migrating WebAssembly components over the Cloud-Edge Continuum

Supervisor
Prof. Tullio Vardanega

Candidate
Massimiliano Baldo

Academic Year 2023-2024

Abstract

With the proliferation of edge computing and the increasing demand for low latency and high throughput applications, the cloud edge continuum has emerged as a promising paradigm for distributed computing. In this continuum, computational tasks are dynamically allocated across cloud data centers and edge devices based on factors such as proximity to data sources, network conditions, and application requirements. However, achieving seamless mobility of application components, especially in the form of WebAssembly (Wasm) modules, presents significant challenges due to differences in hardware architectures, network protocols, and runtime environments between the cloud and edge.

This research explores the concept of live migration of WebAssembly modules within the cloud-edge continuum to enable dynamic resource provisioning and workload balancing. By leveraging containerization technologies and runtime adaptation mechanisms, we propose a proof-of-concept for transparently migrating Wasm modules between cloud data centers and edge nodes without interrupting ongoing computations. The proof-of-concept employs a combination of pre-copy and post-copy migration techniques, coupled with runtime code adaptation, to minimize downtime and ensure data consistency during the migration process.

An evaluation is performed in terms of migration latency and resource utilization. The experimental results demonstrate the feasibility and effectiveness of live migration of Wasm modules in the cloud-edge continuum, highlighting its potential to enhance the scalability, reliability, and agility of edge computing infrastructures. Furthermore, it's discuss open research challenges and future directions for optimizing the migration process, enhancing security and privacy guarantees, and enabling dynamic orchestration of distributed applications in heterogeneous cloud-edge environments.

Contents

List of Figures

List of Tables

Chapter 1

Challenge

This chapter will present a description of the context in which this thesis has been developed. Subsequently, the Cloud-Edge Continuum paradigm will be presented, accompanied by an exposition of its advantages and characteristics. Ultimately, it will be shown that migration presents a significant challenge for this specific paradigm.

1.1 Cloud-Edge Continuum

1.1.1 What is the Cloud computing

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.[9] It has born from the organizations in charge of running data centers. In fact, they have been seeking ways opening up their resources to customers, and eventually, led to the concept by which a customer could upload tasks to a data center and be charged on a per-resource basis. Cloud computing is characterized by an easily usable and accessible pool of virtualized resources [10]. Which and how resources are used can be configured dynamically, providing the basis for scalability: if more work needs to be done, a customer can simply acquire more resources. The link to utility computing is formed by the fact that cloud computing is generally based on a pay-per-use model in which guarantees are offered by customized service-level agreements (SLAs). A section of resources made available by the cloud computing model is posed in Figure 1.1

Referring to Figure 1.1, clouds are organized into four layers:

- Hardware: The lowest layer is formed by the means to manage the necessary hardware: processors, routers, but also power and cooling systems. It is generally implemented at data centers and contains the resources that customers normally never get to see directly.
- Infrastructure: This is an important layer forming the backbone for most cloud computing platforms. It deploys virtualization techniques to provide customers an infrastructure consisting of virtual storage and computing

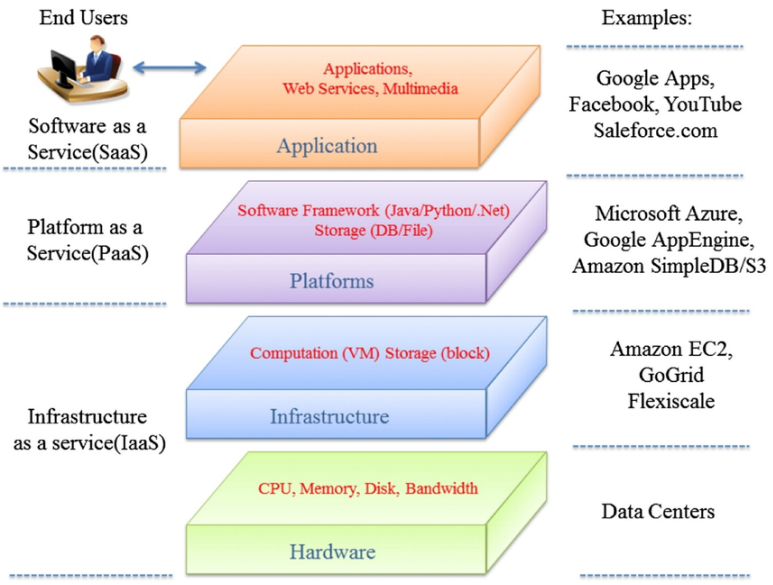

Figure 1.1: Layers of the cloud computing. The left side shows the possible services that a user can take advantage of, while the right one lists the possible examples related to the specific layer [1].

resources. Indeed, nothing is what it seems: cloud computing evolves around allocating and managing virtual storage devices and virtual servers.

- Platform: One could argue that the platform layer provides to a cloud computing customer what an operating system provides to application developers, namely the means to easily develop and deploy applications that need to run in a cloud. In practice, an application developer is offered a vendor-specific API, which includes calls to uploading and executing a program in that vendor's cloud. In a sense, this is comparable to the Unix exec family of system calls, which take an executable file as a parameter and pass it to the operating system to be executed. Furthermore, like operating systems, the platform layer provides higher level abstractions for storage and such. For example, the Amazon S3 storage system[11] is offered to the application developer in the form of an API allowing (locally created) files to be organized and stored in buckets. By storing a file in a bucket, that file is automatically uploaded to the Amazon cloud.

- Application: Actual applications run in this layer and are offered to users for further customization. Well-known examples include those found in office suites (text processors, spreadsheet applications, presentation applications, and so on). It is important to realize that these applications are again executed in the vendor's cloud. As before, they can be compared to the traditional suite of applications that are shipped when installing an operating system.

Cloud-computing providers offer these layers to their customers through various interfaces (including command-line tools, programming interfaces, and Web

interfaces), leading to three different types of services:

- Infrastructure-as-a-Service (IaaS) covering the hardware and infrastructure layer.
- Platform-as-a-Service (PaaS) covering the platform layer.
- Software-as-a-Service (SaaS) in which their applications are covered

Cloud computing as a means for outsourcing local computing infrastructures has become a serious option for many enterprises. From the perspective of a system architecture, which deals with configuring (micro)services across some infrastructure, one may argue that in the case of cloud computing, we are dealing with a highly advanced client-server architecture. However, let it be noted that the actual implementation of a server is generally completely hidden from the client: it is often unclear where the server actually is, and even whether the server is actually implemented in a fully distributed manner (which it often is). To further illustrate this point, the notion of a Function-as-a-Service, or simply Faas, allows a client to execute code without bothering even with starting a server to handle the code.[12]

1.1.2 Properties of Cloud Computing

For a complete view, there will be quickly illustrate the principal advantages and disadvantages about the cloud computing.

It begins by describing its advantages, which are:

- Cost Efficiency: Cloud services are such that one has to pay only for used resources. No buying and maintaining of any hardware upfront is required, thereby supporting this pay-as-you-go mode of operation through economies of scale that result in profit. This is especially true in the case of small startups and small businesses.
- Scalability: Cloud platforms provide enormous scalability. It would allow the firm to create or reduce computing resources as per demand, whether it's peak load or low volume.
- Flexibility and Accessibility: Cloud services allow access to data and applications from any geographical location with an Internet connection. This kind of flexibility enables telecommuting, collaboration, and efficient usage of resources.
- Automated Updates and Maintenance: Cloud providers manage system updates, security patches, and maintenance, easing pressure on IT teams to a great extent. It assures that applications always run on the newest versions without the need for manual intervention.
- Disaster Recovery and Redundancy: Cloud platforms offer inherent redundancy and disaster recovery alternatives. The data is duplicated at different data hubs, so hardware failures and natural disasters will not result in data loss.

Now there will be show its main disadvantages:

- Security Concerns: Storing sensitive information in the cloud is related to security and privacy concerns. Any organization must be very careful while selecting a provider and, in turn implement tight measures of security for their stored data.
- Downtime Risks: Even with very high availability, cloud services could suffer from downtimes in part due to maintenance, outrages, or even cyber-attacks. Businesses have to be prepared for such eventualities through backup arrangements.
- Dependence on Internet Connectivity: Cloud computing requires access to the internet. In case this connectivity is slow or gets disrupted, that impacts productivity and access to other important applications.
- Vendor Lock-In: Migration to another cloud environment is cumbersome and time-consuming. This could lead to relation bondage of an organization with the ecosystem of some vendor, reducing its flexibility.
- Data Transfer Costs: Moving large amounts upstream and downstream of the cloud can be additional expensive. Organizations should consider such charges for data transfer while strategizing in the cloud.

1.1.3 The Edge Computing

In the advent of increasingly more network-connected devices and the emergence of the Internet-of-Things (IoT) and Smart Cities many became aware of the fact that we may need more than just cloud computing. As depicted in Figure 1.2, the number data volume in IoT connection has growth of a factor of 82% from the 2019.

As its name suggests, edge computing deals with the placement of services "at the edge" of the network. This edge is often formed by the boundary between enterprise networks and the actual Internet, for example, as provided by an Internet Service Provider (ISP).

For example, many universities reside on a campus consisting of various buildings, each having their own local network, in turn connected through a campuswide network. As part of the campus, there may be multiple on-premise services for storage, computing, security, lectures, and so on. On-premise means that the local IT department is responsible for hosting those services on servers directly hooked up to the campus network. Much of the traffic related to those services will never leave the campus network, and the network together with its servers and services form a typical edge infrastructure. Concurrently, such servers could be connected to servers of other universities and possibly utilizing, again, other servers. This means that, instead of comparing the universities, there are also configurations where a number of universities share services via a logically centralized infrastructure. This infrastructure may be located in the cloud, or it may have been set up through regional infrastructure making use of locally available data centres. Linked to this move to cloud infrastructures, we often see the term fog computing mentioned.

These may include infrastructures that are critical to monitoring activities, multi-layered video-streaming infrastructures, gaming infrastructures, and so many more. This listing may differ in meaning from one zone to another but

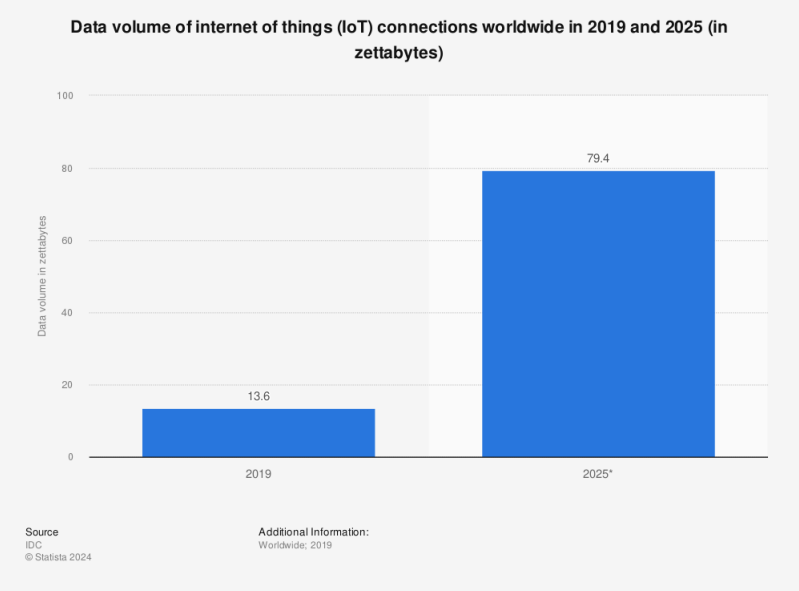

Figure 1.2: Graph representing the amount of data volume in Internet-of-Things connections [2]

has one characteristic in common: its relation to the need for smart end devices to be able to connect either by direct or indirect connection with a service hosted in the cloud. This seeming necessity of an edge infrastructure can lead one to start asking questions about why an edge infrastructure is important. This may appear much simpler at face value—piping directly into the cloud services through established and more reliable networking facilities. But several arguments deserve an in-depth look at why an edge infrastructure is important.

Latency and bandwidth

What should have become clear from our examples is that edge infrastructures are considered to be close to the end devices. Closeness can be measured in terms of latency and often also bandwidth. Throughout the decades, bandwidth, or actually lack of bandwidth, has always been used as an argument for introducing solutions close to specific devices. However, if anything has become clear all this time, is that available bandwidth continues to increase, now reaching the point that one should seriously question how problematic it actually is, and whether installing and maintaining edge infrastructures for having insufficient bandwidth is a good reason. Nevertheless, there are situations in which closeness to end devices is actually needed to guarantee quality of service. The canonical example is formed by video services: the closer the video sources are, the better bandwidth guarantees can be given, reducing issues such as jitter. More problematic is when the video source is far away to the end user. This may easily happen when dealing with latency. It may take 100 ms to reach a cloud, rendering many interactive applications quite useless. One such important application is (semi-)autonomous driving. A car will need to continuously observe its environment through a myriad of sensors and react accordingly. Having to

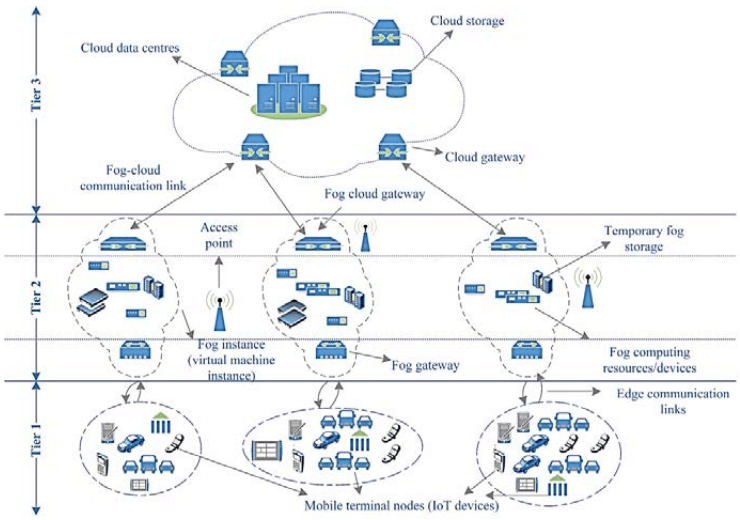

Figure 1.3: Graphical representation of the three types of computation [3]. In Tier 1,there are the devices that are at the "end" of cloud computing, referred to as the edge devices. In Tier 2, there are the devices that act as a conduit for the devices at the edge and the cloud servers. In Tier 3, there are the data-centers that represent the cloud.

coordinate its movements through the cloud is not acceptable from a real-time aspect alone. This example also illustrates that cars may need to detect each other beyond the capabilities of their sensors, for example, when heading toward a junction with clear visibility. In a real-time system, cars may be able to provide their current position to a local edge infrastructure and reveal themselves to each other when approaching the junction. Overcoming latency is one of the most compelling reasons for developing edge infrastructures.

Reliability

Many argue that cloud connectivity is simply not reliable enough for many applications, for which reason edge infrastructures should be deployed. To what extent this is a valid argument remains to be seen. The fact is that for many networked applications, connectivity is generally good and reliable, if not excellent. Of course, there are situations in which relying on 24/7 reliability is not an option. This may be the case for hospitals, factories, and other critical settings in general. Yet, in those cases, measures have been traditionally already taken and to what extent edge computing brings in anything new is not always clear.

Security and privacy

Finally, many argue that edge solutions enhance security and privacy. It all depends One could argue that if a cloud solution is not secure, then there is no reason why an edge solution would be. An implicit assumption that many people make is that an edge infrastructure is owned by a specific organization and operates within the (protected) network boundaries of that organization. In that case, it often does indeed become simpler to protect data and operations, yet

one should ask whether such protection is sufficient. A same reasoning holds for privacy: if we cannot protect personal data in the cloud, then why would an edge infrastructure suffice for privacy? However, there may be another reason related to security and privacy why edge infrastructures are needed. In many cases, organizations are simply not allowed, for whatever regulatory reasons, to place data in the cloud or have data be processed by a cloud service. For example, medical records may have to be kept on premise on certified servers and with strict audit procedures in place. In this case, an organization will have to resort to maintaining an edge infrastructure. Introducing additional layers between end devices and cloud infrastructures opens a whole can of worms compared to the relatively simple situation of just having to deal with cloud computing. For the latter, one can argue that the cloud provider to a considerable extent decides where and how a service is actually implemented. In practice, we will be dealing with a data center in which the (micro)services that make up the entire service are distributed across multiple machines. Matters become more intricate in the case of edge computing. In this case, the client organization will now have to make informed decisions on what to do where. Which services need to be placed on premise on a local edge infrastructure, and which can be moved to the cloud? To what extent does an edge infrastructure offer facilities for virtual resources, akin to the facilities offered in cloud computing? Moreover, where we may be able to assume that computational and storage resources are in abundance when dealing with a cloud, this is not necessarily the case for an edge infrastructure. In practice, the latter simply have less hardware resources available, but often also offer less flexibility in terms of available platforms.

By and large, allocating resources in the case of edge computing appears to be much more challenging in comparison to clouds. The authors Hong and Varghese [13] claim that there are multiple limitations when it comes to resources, higher degrees of hardware heterogeneity, and much more dynamic workloads, which, when taken together, have led to a higher demand of orchestration. Moreover, where from a client's perspective the cloud appears to be hiding many of its internal intricacies, this is necessarily no longer the case, making it much more difficult to do the orchestration [14]. The process of orchestration can be defined based on:

- Resource allocation: specific services require specific resources. The question is then to guarantee the availability of the resources required to perform a service. Typically, resources amount to CPU, storage, memory, and networking facilities.

- Service placement: regardless the availability of resources, it is important to decide when and where to place a service. This is notably relevant for mobile applications, for in that case finding the edge infrastructure that is closest to that application may be crucial. A typical use case is that of video conferencing, for which the encoding is often not done on the mobile device, but at an edge infrastructure. In practice, one needs to decide at which edges the service should be installed [15].

- Edge selection: related to service placement is deciding which edge infrastructure should be used when the service needs to be offered. It may seem logical to use the edge infrastructure closest to the end device, but

all kinds of circumstances may ask for an alternative solution, for example the connectivity of that edge to the cloud provider.

1.1.4 What is the Cloud-Edge Continuum

At the end, the Cloud-Edge Continuum is a paradigm of distributed system in which all the computing models like the cloud, edge, for and mobile are seen as an unique entity, like a continuum.

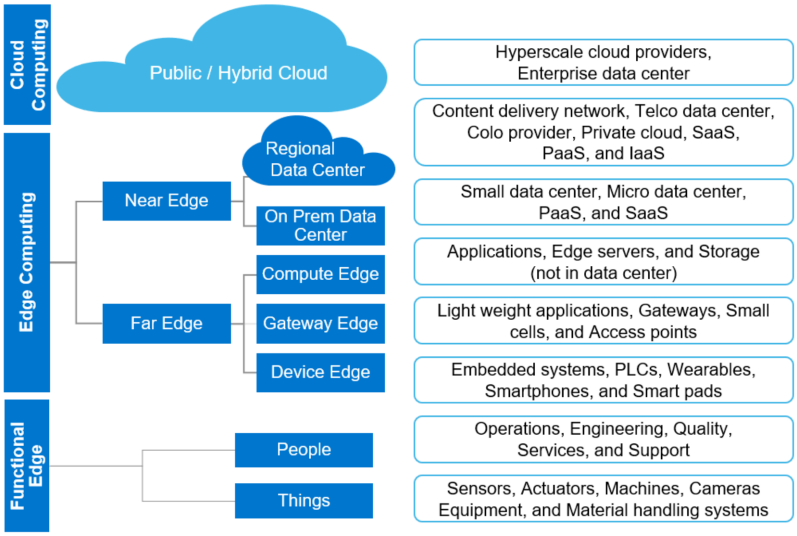

Figure 1.4: Taxonomy of the Cloud-Edge Continuum [4]

Some application areas have been idealized by the "The European Cloud, Edge and IoT Continuum Initiative"[16] for the Cloud-Edge Continuum, some of them include:

- Smart agriculture: precision farming, data autonomy of vehicle swarms, CO2-neutral intelligent farming, smart robots.
- Renewable energy production centres: reliability and security of power grids for smart cities.
- Smart homes: sensors and robots for remote monitoring, enablement of domestic services.
- Factory robotics: reducing CO2 in production, predictive maintenance.
- Logistics and transport: object recognition, energy and process optimisation in movements, predictive maintenance of railway networks, smart ports, last-mile delivery optimisation.
- Healthcare: remote real-time medical diagnostics with machine learning and artificial intelligence-based analytics.

1.1.5 How to achieve it

In order to achieve an implementation of the Cloud-Edge Continuum, it must be possible to "move" from one system to another without manual intervention. As yet, the situation does not allow this to happen, as the different computing models are isolated from one another, creating actual silos. The Figure 1.5 shows graphically what is being described.

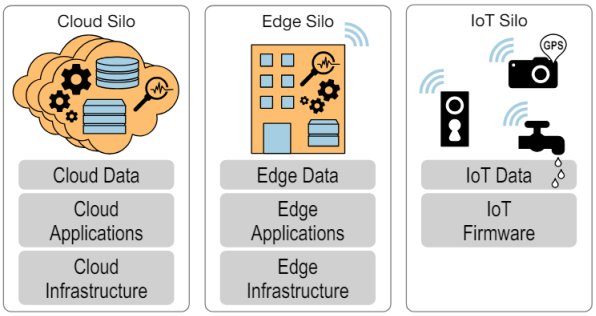

Figure 1.5: Sandboxes of models of computing [5]

As can be seen, the silos in the image have no way of communicating directly with one another, thus forcing the need to operate each of them as individual entities. Therefore, it is necessary to define a mechanism that makes it possible to move from the cloud to the edge and vice versa, and at the same time to find a way so that this move can take place "transparently", i.e. without the user noticing that there has been a move of the application while it is executing. This latter approach is called migration and is presented in the next section.

1.2 Migration

1.2.1 What is migration

In order to explain what is the concept of migration, it's necessary to describe quickly how a distributed system is achieved. One of its important goals id to hide where its processes and resources are physically located across multiple computers, possibly separated by large distances. In other words, it tries to make the distribution of processes and resources transparent, that is, invisible, to end users and applications. Achieving distribution transparency is realized through what is known as middleware[17], which is shown in Figure 1.6.

In essence, what applications get to see is the same interface everywhere, whereas behind that interface, where and how processes and resources are and how they are accessed is kept transparent. The concept of transparency can be applied to several aspects of a distributed system, of which the most important ones are listed in Table 1.1. It has been used term object to mean either a process or a resource.

An important group of transparency types concerns the location of a process or resource. Location transparency refers to the fact that users cannot tell where an object is physically located in the system. Naming plays an important

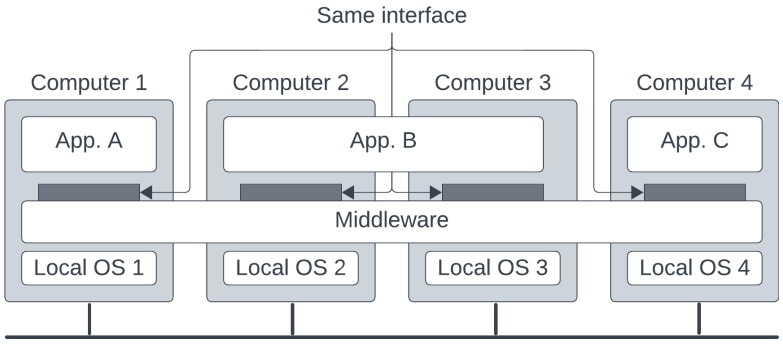

Figure 1.6: A middleware layer which offers the same interfaces to all the connect machines

Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Table 1.1: Different forms of transparency in a distributed system.

role in achieving location transparency. In particular, location transparency can often be achieved by assigning only logical names to resources, that is, names in which the location of a resource is not secretly encoded. An example of a such a name is the uniform resource locator (URL) https://www.unipd.it/, which gives no clue about the actual location of the Web server where this book is offered. The URL also gives no clue whether files at that site have always been at their current location or were recently moved there. For example, the entire site may have been moved from one data center to another, yet users should not notice. The latter is an example of relocation transparency, which is becoming increasingly important in the context of cloud computing: the phenomenon by which services are provided by huge collections of remote servers. Where relocation transparency refers to being moved by the distributed system, migration transparency is offered by a distributed system when it supports the mobility of processes and resources initiated by users, without affecting ongoing communication and operations. A typical example is communication between mobile phones: regardless whether two people re actually moving, mobile phones will allow them to continue their conversation. Other examples that come to mind include online tracking and tracing of goods as they are being transported

from one place to another, and teleconferencing (partly) using devices that are equipped with mobile Internet

An actual implementation of the technique of migration is used by the hypervisor for creating virtual machines. Figure 1.7 shows a graphical representation of ones.

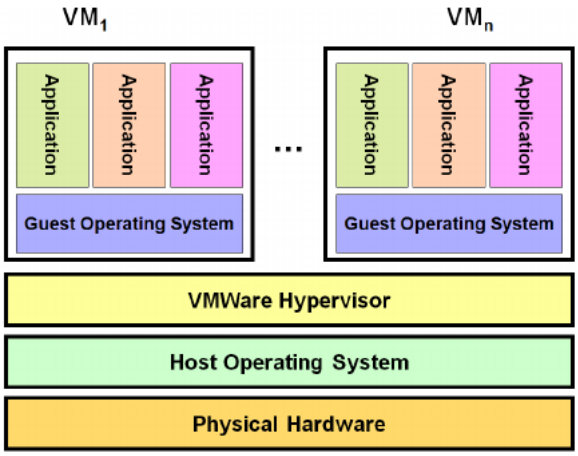

Figure 1.7: Depiction of a Virtual Machine inside a host [6]

A virtual machine is a fully feature computer which is digitalized inside another computer, i.e is a process in which is simulated all the hardware competent to run an entire operating system [18] [19]. This kind of VM[1], which are called "guests", are totally isolated from the machine that hosts them, which is called "host". For this reason, it's possible to have multiple virtual machines that are executing at the same time with different OS from each other. To support this technology, there must by an hypervisor which is the bridge between the guest and the host machine. It's task is to provide all the necessary virtualized resource to the guest machine, by requesting to host to provide them. Since the entire OS is virtualized by the host, it's possible to execute a migration from one host to another. To do so, in the next section it will be described different approaches for achieve a migration mechanism.

1.2.2 Type of migrations

Now there will be presented different typologies of migration techniques. There will used the virtual machines as resources to migrate, but the mechanism can be abstracted and use to any kind of resource, like for example an program execution. The Figure 1.8 shows a scheme for the characterises of each method. One approach is the so called clod migration (figure a of Figure 1.8), in which the virtual machine is shut down and all its resources are copied to the new host to be instantiated. Its the easiest way to implement an mechanism of migration, although it generates a desecrate downtime, i.e. a time of disservice in which the user can't use the virtual machine.

A more interesting one is the hot migration, also called as the live migration. This technique is transparent for the user, in other words latter one continues

[1] Abbreviation for Virtual Machine

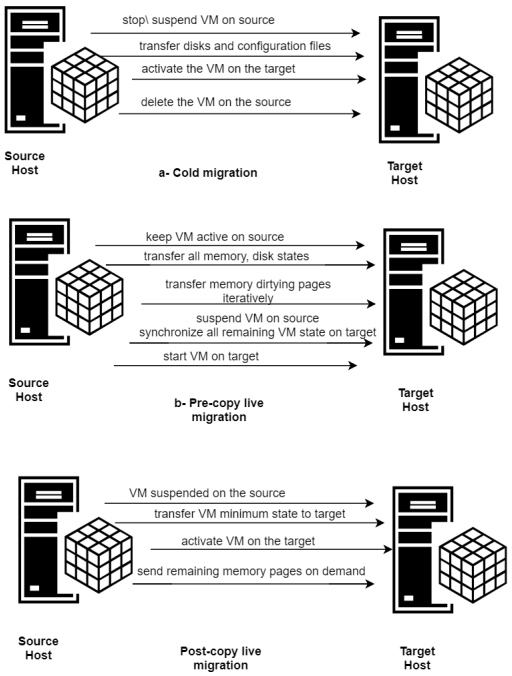

Figure 1.8: Depiction of possible methodologies for migrate a virtual machine [7]

to utilize the VM without noticing anything. Behind the scenes, the hypervisor generates a snapshot, a data representation of the state of the virtual machine in a specific moment. Then, it sends it to the new host for be instantiated and continues its execution. Even with this approach, there are different techniques that can be adopted, which will be discussed in the next section. The problem related to the live migration consists in how to handle the modification of the VM during the migration. This issue is not present in the cold migration since the user can't use the VM, and so when the latter is shut down is state is constant. Instead, in this case users are still able to interact with the virtual machine, and so modify its state. There exist different types of hot migration based on how they handle this issue. The following categorization is based on the work of Alhammadi [20], which defines the differences in a detail way. There are three different approaches for implementing the live migration:

- Post Copy: here the virtual machine is first quickly paused; then, a basic processor state is copied, and then the virtual machine is rebooted at the destination. Later, each memory page needed by running applications is fetched from the source (figure b of Figure 1.8).
- Pre-copy: in this case, the memory state is migrated and next the CPU state is migrated. There are two stages in this pre-copy method, which are the Warm-up stage and the Stop-and-Copy stage. In the Warm-up stage, the hypervisor just copies all the pages of main memory of the VM from the source to destination and allows the VM to continuously keep running

on the source without any kind of interruption. This dirties memory pages; if any memory pages get modified during the memory copy, dirty pages are created, having to be copied again. During the Stop-and-Copy phase, the source VM will be paused, the remaining dirty pages will be copied to the destination, then it will restart at the destination (figure c of Figure 1.8).

- Hybrid: The hybrid algorithm was a good option to recover the limitation between the pre-copy algorithm, towards the post-copy algorithm, and vice versa. The main goal was to reduce the amount of page fault as few as possible, while ensuring the time of migration was kept constant.

1.2.3 How to achieve it in the Cloud-Edge Continuum

As outlined in the section 1.1, to create a unique layer composed by the cloud and edge computing its necessary to be able to make recourse pooling even at the edge of the network. For this reason, a migration mechanism is essential to obtain a first step in the implementation of the Cloud-Edge Continuum. Due to the heterogeneity of devices which composed the edge, a possible solution would be to create a common layer which both cloud server and edge device can support. The authors Ménétrey et al.[5] have proposed the technology WebAssembly as a common layer for the execution of application in the cloud-edge computing. Moreover, they depicted how studying the migration an execution of WebAssembly module outside of the web would be an interesting research. In the next section, will be presenting this new kind of technology, showing that its characteristics make it a valid candidate for being execution platform of the Cloud-Edge Continuum.

1.3 WebAssembly

1.3.1 What is WebAssembly

WebAssembly is an Instruction Set Architecture for a stack-based virtual machine. [21] The first motivation behind WebAssembly is to exploit the heterogeneity of programming languages defining a common compilation target. In fact, before WASM[2], the only way of creating a web application was to use JavaScript. The reason is that the rendering engine present in the browser can interpret only JavaScript code, leveraging it as the only programming language for the Web. So, instead of creating a new programming language that eventually will be become like JavaScript, the authors idealize a technology complementary to JavaScript. In fact, once the logic of the application is moved inside WebAssembly, JavaScript would be used only to fetch and load the module in order to be executed. This type of interaction is called “glue called”, since JavaScript is the glue that wire the virtual machine inside the render engine and the WebAssembly module.

Since WebAssembly is a low-level target, one of its peculiarity is the linear memory, which is a contiguous untyped slice of bytes similar to an array [22]. Inside the linear memory there are all the necessary information to execute a WASM module. A simple representation is depicted in Figure 1.9.

[2]Abbreviation for WebAssembly

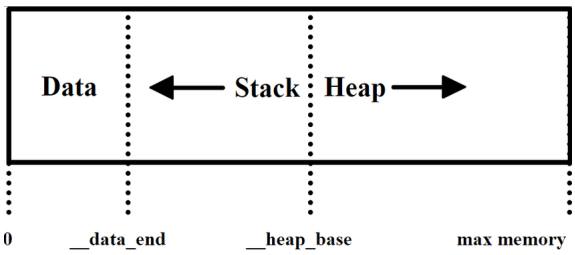

Figure 1.9: Simple graphical representation of the linear memory of a WASM module [8]

This feature is so important in WebAssembly since it's the only way the external JavaScript code and WebAssembly can communicate together. An example of usage is represent in Listing 1.1.

```
1  WebAssembly.instantiateStreaming(fetch("memory.wasm"), {
2      js: { mem: memory },
3  }).then((results) => {
4      const summands = new DataView(memory.buffer);
5      for (let i = 0; i < 10; i++) {
6          summands.setUint32(i * 4, i, true);
7      }
8      const sum = results.
9                  instance.
10                 exports.
11                 accumulate(0, 10);
12     console.log(sum);
13 });
```

Listing 1.1: Snippet of JavaScript code for accessing the Linear Memory of a WASM module

The first three lines are the glue code for loading the WebAssembly module and then executed. In line 2, there is a JSON object that represents the names with JavaScript will access the WebAssembly's resources. So in this case, the variable "memory" of JavaScript will be associated with the variable "mem" of WebAssembly, which it's the linear memory. After the "then" keyword, there is the resolution of the promise of loading the module. The variable "summands" represents the actual content of the linear memory, which at the begging is empty. Line 5 and 6 fill the memory with the function "setUnit32" which sets an integer value (represented in 32 bits) in the "i" position. The last parameter if for explicit use the little endian format. Lastly, at line 9 there is a call to a function which is not defined in the block of code in Listing 1.1. In fact, that function "accumulate" is defined inside WebAssembly and it's called outside of it. That function sums all the value that is stored in a specific range of indexes of the linear memory.

Now that has been shown the functionally of WebAssembly, it will be discuss its two main advantages.

1.3.2 Characteristics of WebAssembly

Portability

The concepts of portability is concept is antithetical to the notion of polyglot architectures and applications, which afford developers the flexibility to utilise the language of their choosing. In fact, it's possible to target WebAssembly from different programming languages like Rust, Go, C and so on. This results in a reduction in the amount of work required, an increase in the reuse of code, an improvement in security, and a significant enhancement of the developer experience. Given that the concept of WASM was founded upon the principles of flexibility and polyglot capabilities, it is evident that portability is inextricably linked to the expectations inherent to the WASM ethos. The portability that was a significant selling point of containers in the context of cloud computing. Although, it has bee demonstrated that WebAssembly has reaches better performance respect to container technologies [23] [24] [25].

Security

Another characteristic of WebAssembly is the *security*. WebAssembly code runs closed into a sandbox managed by the vm of the browser's engine or the runtime, in any case for sure not by the operating system. This gives it no visibility of the host computer, or ways to interact directly with it. Access to system resources, be they files, hardware or internet connections, can only happen by explicating which resources can assess that module. This philosophy is called Principle of least privilege [26], which is used in the UNIX based operating systems. Moreover, compared to normal compiled programs, WebAssembly applications have very restricted access to memory, and to themselves too. WebAssembly code cannot directly access functions or variables that are not yet called, jump to arbitrary addresses or execute data in memory as bytecode instructions. Inside browsers, a WASM module only gets the linear memory which has been described previously. WebAssembly's module can directly read and write any location in that area, or request an increase in its size, but that's all. The linear memory is also separated from the areas that contain its actual code, execution stack, and of course the virtual machine that runs WebAssembly. For browsers, all these data structures are ordinary JavaScript objects, insulated from all the others using standard procedures.

1.3.3 Why using it outside of web

Due to its versatility, many people have seen that it can be an advantage to use WebAssembly outside of the browser, particularly in the context of Serverless computing. The Serverless computing, called also as Funtion-as-a-Service, is an execution model of the cloud computing, and its main benefits realise on not worrying about the configuration and management of the resources to execute an application [27]. It has been seen how the main technology used (which is the Docker container) for dispose the execution of multiple applications, can have bad performers in constrained environment like IoT device [28]. For this reason, it has been proposed WebAssembly as new layer of execution for FaaS[3],

[3]Abbrevation for Function-as-a-Service

obtaining better cold-starts and resource usage [29] [30]. In order to achieve this technology, it's necessary to dispose of a runtime which simulates a VM browser engine. Along with the runtime, the creator of WebAssembly has defined a standard for the system call that can be used inside a WASM module. This set is called WASI, which stands for WebAssembly System Interface[31]. Its nature is similar to POSIX, which is used in all the UNIX based operating systems. This kind of specifications allows to use any WebAssembly runtime which supports WASI, no matter what is the real implementation of the system call.

The context in which this elaborate takes place has been presented. The initial stage of the presentation delineated the concept of cloud computing, subsequently introducing the supplementary notion of edge computing. Subsequently, the Cloud-Edge Continuum was introduced, accompanied by an exposition of its defining characteristics and applications. The necessity of the migration property for the attainment of this paradigm has been elucidated, as well as the means of achieving it. In conclusion, WebAssembly has been proposed as a potential candidate for use as a common layer through which to migrate the execution of applications within the Cloud-Edge Continuum.

Chapter 2

Vision

The following chapter will present the project's vision and the desired outcome. The chapter will present an overview of the state-of-the-art technologies related to WebAssembly runtimes, and will finally set out the targets for this elaborate.

2.1 Ideal architecture

The vision of this project is a focus on a doctoral research topic, which is a wider and more complex work. For this reason, an overview of that theme can better frame the nature of this work. In the doctorate investigation, the aim is to create functional runtime for the cloud-edge continuum in which the computation, and its data, move forwardly from the cloud to the edge of the network, and vice versa. The motivations for why a computation must be moved can be various, an example would be that the execution must be as near as possible to the source of the request. The main characteristic of this runtime would be that this "movement" of the computation can be predicted by the devices that from the Cloud-Edge Continuum. As shown in Section 1.2.1, and more precisely on Table 1.1, to obtain a system like this one there must be some degree of transparency. This property is composed of different aspects, from how the user accesses the data to how hides the failure of executions. One of these is the migration, the ability to "move" the resources or process without showing to the user. So the vision for this work is to implement a mechanism to migrate an executive from a host to another without stopping it.

To achieve this aim, it will be describe the possible architecture for this system. The final result would be a system consisting of several members, defined "nodes". Since the context is the cloud-edge continuum, these nodes can be a mainframe or an Internet-of-Things webcam for surveillance, as an example. So, the fact to take in mind is that these nodes are heterogeneous and support portability, i.e. they can change places. As discussed in Section 1.3, WebAssembly is an optimal candidate to be used as common layer of execution for these devices. Therefore, these nodes must be able both to instantiate and run WebAssembly modules. Thus, a WebAssembly runtime must be installed on all these devices.

Next, there is the requirement to implement a live migration mechanism for WASM modules. In Section 1.2.2, different types of live migration approaches

have been presented. All of them require a generation of a "snapshots" that represent the state of an execution in a specific moment. Also, there must be a mechanism to restore an execution (of a WASM module) starting from a snapshot. Thereby, the runtime must be able to do both the tasks. In addition, the node must be able to send a snapshot to other nodes, and be able to receive snapshot copies from other nodes.

Subsequently, a solution for receiving and sending snapshots inside the nodes of the system is needed. It cannot be the runtime to do since it has already to manage the lifetime of the WASM module, so there must be a web server (which is even a client) install on the nodes. It will be the bridge of communication between the runtime and the outside world of the node. Since there is the presence of portable devices, resources used by a WASM module can be lost during the phase of migration. It would be necessary to consider a mechanism of re-connection for the WebAssembly module after being migrated to another host. In addition, a node cannot know where precisely sends a snapshot because it doesn't have a uniform vision of all the system. Hence, there must be a handle to coordinate the interchanges of nodes inside the system.

Considering all the requirements, the image 2.1 shows a possible a representation of the system.

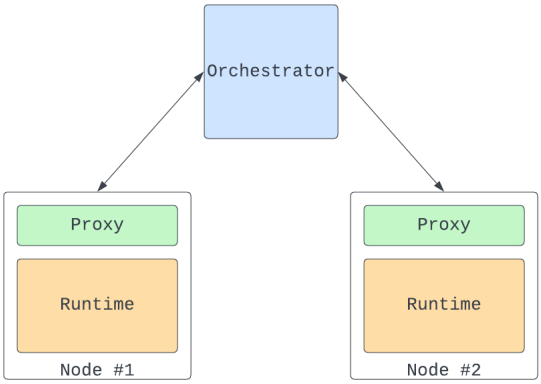

Figure 2.1: Graphical representation of the ideal system

The components for the realization of the system are the following:

- *Runtime*: formally speaking, it is not a runtime but an execution environment. This component, in addition to the classic tasks of instantiation and execution of WASM modules, must be able to generate snapshots and resume execution from them. Since there are nodes within the system with limited resources, i.e. edge devices, the component must have a size optimized for such scenarios;
- *Proxy*: the proxy component handles incoming and outgoing internet connections from the node. Think of a web server API, which exposes routes to get a snapshot of a WASM module or request a run restore. This component will interact directly with the "runtime" to meet requests. In addition, being the mobile end user you need to use a connection recovery mechanism between the user and the new host;

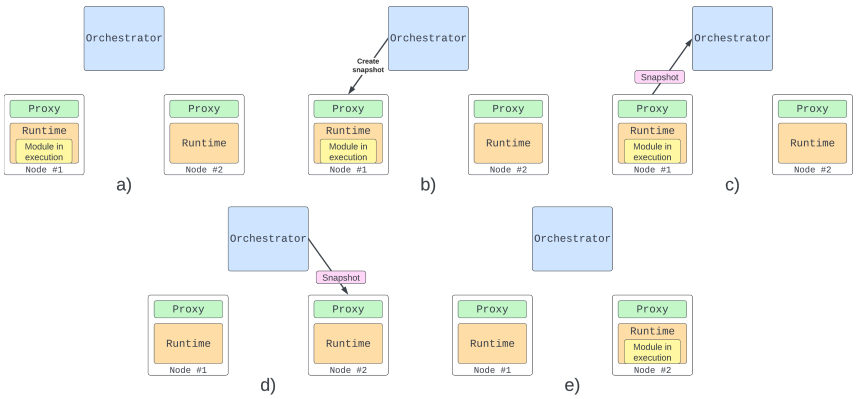

Figure 2.2: Graphic representation of process migration

- *Orchestrator*: this component has the task of initializing the migration process within the system. Based on external events, such as the physical move of the client, the orchestrator will find the last snapshot of the running module requested by the client, determine the new node to move the execution to, and then request the re-installation of the module.

To better show the interactions and roles of the individual components, we describe the migration process idealized for this system. The Image 2.2 shows this process graphically.

The migration process is divided into the following steps:

a) At the beginning, there are two (or more) different nodes in which there are installed, the *runtime* and the *proxy* for execute WASM module. In this example, Node #1 is executing the actual WASM module;

b) At some point, the orchestrator decides to start the migration. To do so, it requests the hosting node, in this case Node #1, to generate the snapshot of the actual execution;

c) Node #1 sends the snapshot to the orchestrator, the latter one will decide which will be the new hosting node;

d) One the orchestrator finds a valid candidate, it request for this one to restore the execution from the snapshot. In this example, the candidate is Node #2;

e) At the end, Node #2 starts executing from the given snapshot and Node #1 stops its execution.

2.2 State of the art

Some solutions have been proposed for the migration of WebAssembly modules, but no one satisfied the vision of the project. The researches [32] [33] showed how to leverage trusted execution environment, also called TEE, to run WebAssembly modules and build a migration mechanism on them imitating the

resources which the module can assess. These solutions require specific hardware and software to use this technology, which in the context of heterogeneity it's hard to satisfy. Moving inside the web, in the work [34] have idealized a solution to migrate WebAssembly modules present on the web through a system of asynchronous messages supported by JavaScript worker threads. A novel runtime for WebAssembly called Lunatic[35] integrates the primitive for communicating with other instance via messages, like the actor model in the Erlang VM[36]. However, there is no support for migrating the execution of a WASM module. Finally, there is a migration tool called CRIU [37] that allows you to migrate UNIX processes using the checkpoint/restore technique. The latter requires that both machines on which the checkpoint/restart is run have the same architecture.

2.3 Objectives

Will be now presented the objective set for this work. The actual state of the art doesn't propose any kind of solution for obtaining an WASM runtime with support to migrate the computation modules. However, the principal steps to obtain a solution like that are the following:

1. access and handle the state of the module during runtime;
2. replicate it outside of the execution;
3. restore the execution of the same module but starting from a saved state:

These steps represent the distance obtain an initial solution for the system presented in 2.1. So, the main objective of this work is to complete the steps and obtain a proof of concept for a WebAssembly runtime that can migrate module's execution. Then, as secondary goals, it has set to investigate and integrate a solution for handle re-connections inside the system. More specifically, defining a solution for:

1. support client re-connection;
2. support server re-connection, that is, open connections from the WASM module.

The system's vision is primarily constituted by three key components: a novel WebAssembly runtime, a web server, and an orchestrator. Subsequently, it has been demonstrated that no solution has been proposed for the implementation of these components in order to obtain the desired system. In light of the aforementioned considerations, a series of objectives have been established with the aim of substantiating the viability of this system.

Chapter 3

Solution

This chapter presents a feasible solution, accompanied by a detailed description. The following section presents a number of key features of the implementation, together with their respective meanings. Finally, in order to ascertain the quality of the work, a series of experiments have been designed and their outcomes presented and discussed.

3.1 General idea

3.1.1 First attempt

During the phase of idealization for a proof of concept, an initial attempt was to reinitialize the execution of the WASM module and restoring its linear memory. This idea came from [34] in which the authors do the same approach but inside the web browser. The problem of this solution consisted in the lack of information related to which operation was executed before the migration. Indeed, even if the linear memory of the module was restored, the execution was not maintained during the migration. This caused an initialization from zero, i.e. without any record of the previous execution.

Since one of the objects of this work was to obtain a migration mechanism without the need of initialization, this solution was rejected. Despite this conclusion, the hours employed in this first attempt allowed a better knowledge of how WebAssembly module is executed.

3.1.2 Proposed solution

The solution presented is based on the concept of checkpoint/restore. This technique is already used in containers through the use of CRIU [37], but also in real-time systems where you are not sure that the execution always arrives at the end. The checkpoint consists of an intermediate saving of the execution state, and this saving is generated with a certain frequency. While the restore consists in recovering the execution from a state within the checkpoint, thus avoiding the need to have to re-enter from scratch. In order for this technique to be used, idiomatic programming must be introduced. It means a set of rules that, if respected allow the program to have certain properties, in this case it means the possibility of migrating the execution. What is required by

the WASM module is the use of the two checkpoint/restore functions to save the execution state. At this point, the problem arises that WebAssembly does not have any instructions either to save the state or to determine the state of execution. To solve these problems we can consider that:

- you can import functions external to the WebAssembly module and use them within the module. Therefore, by modifying the execution environment to implement the checkpoint/restore functionality, it would be enough to import the signatures of the functions;
- WebAssembly uses a contiguous memory called linear memory, in which all the information and data of the module are saved. If you represent the state of execution in the form of a vector, and of this you know the location (in memory), you could access that address and read the content, or even change it.

Therefore, the required idiomatic programming consists in:

1. represent the logic of the program in the form of a vector in which each position corresponds to a variable;
2. import the two signatures of the checkpoint/restore functions, using once the restore (since it only takes once to recover the previous state) and the checkpoint as many times as the state, that is the vector, is changed.

You thus get a WASM module that can be run without the need to start from scratch. When editing the execution environment, you should also consider the possible migration request and how the migration affects the snapshot creation. Consider the time t at which the checkpoint and s at which the migration request occurs, there are three possible cases:

- The request is made before the checkpoint, ie $s < t$, therefore the status is retrieved at time $t - 1$ and re-match the instructions before s;
- The request takes place during the checkpoint, that is $t = s$, in this case if the migration happened it would create an inconsistent state. Therefore, the migration phase is suspended until the checkpoint is completed;
- The request occurs after the checkpoint, ie $s > t$, and then you can use the time t state.

3.2 Proposed Architecture

3.2.1 Runtime

The Wasmtime library [38] was used to implement the runtime execution environment. The latter is a standalone implementation of WebAssembly written in Rust, designed to be embedded in applications and environments that require WebAssembly code execution. It is primarily developed by the Bytecode Alliance team, which includes members such as Mozilla, Fastly, Red Hat and others. Wasmtime provides an API that allows developers to easily integrate WebAssembly code execution into their applications. It can be used in a wide

range of scenarios, including servers, embedded devices, programming language runtime environments, and more. One of the key features of Wasmtime is its flexibility and modularity. It is designed to be highly configurable and can be extended with additional functionality through the use of external plugins and libraries. As Wasmtime implements the WebAssembly specification, a simple configuration to run a WASM module requires the use of several modules and data structures. Therefore, to simplify the process of creating and running a WASM module, it was decided to create two data structures: the Environment and the Snapshot. The first represents the life cycle of a WASM module, starting from its compilation until its termination, acting as an abstraction for the underlying components. The last represents the state of execution of a WASM module at a certain moment. An Environment is associated with a Snapshot, which is overwritten whenever the checkpoint function is invoked in the WASM module. Next, we defined the implementation of the checkpoint and restore functions, which will then be linked to the WASM module at compile time. To achieve this, one of the elements of Wasmtime technology was used, namely the Linker. As you can see from the name, the Linker has the task of connecting implementations of external functions to the WASM module to the same module.

The two implementations of the functions are now shown:

```
1  linker.func_wrap(
2      "migration",
3      "checkpoint",
4      |mut caller: Caller<'_, Arc<Mutex<Snapshot>>>,
5      ptr: i32, len: i32| {
6          //Access to linear memory of module
7          let memory = match caller.get_export("memory") {
8              Some(Extern::Memory(mem)) => mem,
9              _ => anyhow::
10                 bail!("failed to find host memory"),
11         };
12
13         // Block the mutex in order to prevent migration
14         // during the reading of linear memory
15         let mut state = caller.data().lock().unwrap();
16         // Define an empty slice of bytes
17         let mut data = vec![0u8;
18                             len as u32 as usize *
19                             size_of::<i32>()];
20         // Read the content of the linear memory at
21         // pointer ptr and saved the content inside data
22         memory.read(&caller,
23                     ptr as u32 as usize,
24                     &mut data)?;
25
26         // Save the content inside
27         // the state of the runtime
28         state.section = data;
29         // Set the control variable to restore
```

```
30          // a state in case of migration
31          state.restore = true;
32
33          Ok(())
34      },
35  )?;
```

Listing 3.1: Implementation of checkpoint function

The function 3.2.1 represents the implementation of the checkpoint function. The logic behind this code is to access a specific portion of memory, in which there is the vector containing the state of the WASM module, copy it's content and saved inside a snapshot. What's challenging to understand is in line 4, the *caller* variable. It represents the module that has called the function, but to that module is attached a type Arc⟨Mutex⟨Snapshot⟩⟩. The Arc⟨Mutex⟨⟩⟩constructor is used in Rust for enable safe concurrency for shared state variables. The Arc means *an atomically reference counted* type, basically it's possible to use the type inside the Arc in different context, like for example threads. The Mutex is the classical mutex, i.e. a section that only one thread at time can access and interact with it. At the end, the Snapshot is the data structure described before. Using this pattern, when the function checkpoint it's called from the module then none can access to the content of the Snapshot until it had done. This satisfies the property to not create inconsistent snapshot during migration.

```
1  linker.func_wrap(
2      "migration",
3      "restore",
4      |mut caller: Caller<'_, Arc<Mutex<Snapshot>>>,
5      ptr: i32, _len: i32| {
6          //Access to linear memory of module
7          let memory = match caller.get_export("memory") {
8              Some(Extern::Memory(mem)) => mem,
9              _ => anyhow::
10                 bail!("failed to find host memory"),
11         };
12
13         // Accessing to data of snapshot for restoring
14         // the execution
15         let state = caller.data().lock().unwrap();
16         let restore = state.restore.clone();
17         let section = state.section.clone();
18         // Exiting the shared state content
19         drop(state);
20
21         // Check if there is the need to restore
22         if restore {
23             // Write the previours content
24             // in the linear memory
25             memory.write(caller,
26                          ptr as u32 as usize,
```

```
27                          &section)?;
28                println!("RESTORED!");
29            }
30
31            Ok(())
32        },
33    )?;
```

Listing 3.2: Implementation of restore function

The same reasons is applied to the function 3.2.1. The only differences is that here there is a check for restoring or not, since when a new module is instantiate it doesn't need to be restore since it's never been executed. As opposite of the checkpoint, the content of the snapshot is written where is stored the vector in the linear memory, so the state has been recovered.

3.2.2 Idiomatic Programming

Transitioning to the utilization of checkpoint/restore functions, a potential application is described through a subsequent useful example. An example of idiomatic programming is provided as described in the section. To illustrate, let's consider the implementation of a function, fib, which, given a number n, returns the Fibonacci sequence value at position n. As WebAssembly serves as a compilation target, any language supporting such compilation directives can be utilized. In this example, Rust is employed as the source code. Below is the implementation resolving the problem:

```
#[link(wasm_import_module = "migration")]
extern "C" {
    fn checkpoint(ptr: *mut i32, len: usize);
    fn restore(ptr: *mut i32, len: usize);
}

#[no_mangle]
pub fn fib(n: i32) -> i32 {
    let mut state = vec![0, 1, 1];
    unsafe { restore(state.as_mut_ptr(), state.len()) };

    if n == 0 {
        0
    } else if n == 1 {
        1
    } else {
        while state[2] < n {
            let result = state[0] + state[1];
            state[0] = state[1];
            state[1] = result;
            state[2] += 1;
            unsafe {
                let mut new_state = state.clone();
                checkpoint(new_state.as_mut_ptr(),
```

```
                        new_state.len())
                };
            }
            state[1]
        }
}
```

The first step involves importing the signatures of the two functions, checkpoint and restore. Note how the "no_mangle" attribute instructs the compiler not to modify the original names of the functions; otherwise, there would be a function call issue. Subsequently, the implementation of the fib function is provided. Instead of using local variables, a vector containing the Fibonacci function value at position minus one, at position minus two, and the current index is utilized. This vector represents the execution state of the module. Following this, the restore function is called, which initializes the function to a previously saved state in case of migration. If it's a zero instantiation, default values are used. The function takes the pointer to the vector and its length as arguments so that the execution environment can read the contents of the vector within the linear memory. At each iteration, the vector is modified with new values necessary to compute the result. Every modification of the vector represents a change of state within the application; hence it's advisable to save this state. The checkpoint function, using the same arguments as the restore function, is employed for this purpose. The computation result is returned once the value of n is reached.

3.2.3 Proxy

Now will be describe the last component needed for having a functional migration mechanism, that is the proxy. As anticipated in Section 3.2, it is a wrapper for the underlying modified runtime and it's duty is to handle HTTP communication for serving snapshot of the actual WASM module and restoring an execution starting from a snapshot. To achieve this, it has used the library Axum [39] which simplifies the creation of HTTP web server. Since concurrent access to the same resource would generate data races, it has been decide to wrap the state of the application around an Arc⟨Mutex⟨⟩⟩construct. In this way, only one request at the time would be served.

There have been implemented two handlers, one for the snapshot and one for the resume. These handlers are functions that will be called when a specific rout would be request buy some client. Starting with the former, its implementation is presented in Listing 3.2.3.

```
1  /// Handler for getting the Snapshot
2  /// of the WASM module, if there is one
3  pub async fn snapshot(State(state): State<AppState>)
4  -> Result<Json<Snapshot>, StatusCode> {
5      let _data = state.
6                  data.
7                  lock().
8                  expect("mutex was poisoned");
9      match _data.clone() {
10         Some(env) => Ok(Json(env.snapshot())),
```

```
11          None => {
12              eprintln!("Trying to get snapshot \\
13                          of non-existing porcess");
14              Err(StatusCode::NOT_FOUND)
15          }
16      }
17  }
```

Listing 3.3: Implementation of the handler for serving snapshot

In the first line there is the access to the mutex, and to do so is request the acquisition of the lock. Then, the code is pretty straight since there is a check if a module is actually executed. If so, it will return its snapshot otherwise there will be an error since there is no module in execution. The snapshot is in a JSON format, so it is much easier to work around.

Moving to the second handler, the Listing 3.2.3 represents its implementation.

```
1  /// Handler for starting the execution
2  /// of a given Snapshot in form of json object
3  pub async fn start(State(state): State<AppState>,
4              Json(payload): extract::Json<Snapshot>) {
5      let _env = Environment::from_snapshot(payload,
6                          String::from("fib")).unwrap();
7      let mut _data = state.
8                      data.
9                      lock().
10                     expect("mutex was poisoned");
11     *_data = Some(_env.clone());
12     drop(_data);
13     let _ = _env.execute();
14 }
```

Listing 3.4: Implementation of the handler for resuming an execution from snapshot

In this case, a snapshot is given as argument and it's request to resume an execution starting from it. In the first line it's invoke the function that create a new environment, in which it would be copied the content of the array in the linear memory. Then a reference of the new environment is saved in the application state, so from now on it's known that there is an execution of a module. Lastly, the new module would execute starting from the restore function and continuing its execution.

3.3 Experiments

To make an object evaluation of this project, it has been idealized some experiments. As anticipated in Section 2.2, the actual state of the art no WebAssembly runtime that can migrate the module's execution. Therefore, it is not possible to objectively compare the solution with others, since they do not exist. So this there is no possibility to compare with other solutions, what have been

done. Perhaps, since the project has been created building upon an existing WebAssembly runtime, a way could be to compare the new WebAssembly runtime with a "naive" version of it, which represents the state of the art. At this point, finding the differences of the "naive" and "new" runtime in term of time and memory can determine if the implemented mechanism worsens the actual runtime performance. In the end, to give an objective evaluation to this solution, the following three questions have been idealized:

1. Does the idiomatic programming required to create migratable WASM module take up more memory than a native solution, or does it have overhead? If so how much?

2. How do the checkpoint/restore functions affect the instantiation and execution of a WASM module?

3. How long does it take to checkpoint and restore? Is this cost fixed or variable?

The expectations about the resolution of this experiment are that there will be an overhead introduce by importing the functions of the checkpoint and restore, but it will be small. About the cost of calling the functions, during instantiation will be absent since at least there will be one call to restore the array of values (the state of execution), while during execution, the checkpoint will influence the overall time since there will be multiple calls to save the state. Finally, to show that the cost of the functions is fixed, it has to show for all the possible programs that can be implemented with the idiomatic solution. Moreover, more information is stored in the array in the linear memory more time will be taken to restore the state. So at the end the assumption is that the cost of the functions changes based of the implemented application.

Now follows the configuration and analysis of the results of these experiments.

3.4 Results and evaluations

These experiments represented the answer introduced in Section 3.3. Since there are request metrics about the life cycle of a WASM module (like instantion and execution), there must be provided a WASM module. Out of convenience it has been using the example module described in Section 3.2.2, which is the implementation of the Fibonacci algorithm. The approach was to set a fixed argument that would generate a huge output, taking more time respect a lower one, with the goal to have time measurements in the order of milliseconds. The ideal technique would be to idealize a suite of specific modules for the proposed solution, since inferring from one only can be too strict, but it was not the core of this project so it has been put aside for a future work.

Lastly, the experiments have been run on a server with an Intel i5-6400 quad-core, four threads at 2.7 GHz; 16 GB of RAM and a Ubuntu 22.04 LTS (Jammy Jellyfish).

Native	Idiomatic
1442133	1504129
Overhead 4.12%	

Table 3.1: Comparison the size of the WASM module in native code and in idiomatic code.

3.4.1 First experiment

To determine the presence of a possible overhead, it's necessary to find the effective measurements, i.e. what to measure. In this case, the key is the amount of memory used to store the WebAssembly module, with and without using the idiomatic programming style. So, using the example module explained in Section 3.2.2, there are created two different files respectively called "fib-before.wasm" and "fib-after.wasm". The former will be written using the classic style, while the latter using the idiomatic style proposed in section 3.1. At the end, the command "stat" displays some file's status, even the actual bits used by a file. In Figure 3.1 is depicted the result of that command applied to the two files.

(a) Status of "fib-before" module **(b)** Status of "fib-after" module

Figure 3.1: Status of the two version of Fibonacci's module

The interested filed is the "size", which for the file "fib-before.wasm" is about 1442133 bits and for "fib-after.wasm" is about 1504129. At first glance, there is some overhead that has been introduced using the idiomatic style. To quantify it, it must be used the following formula $overhad = difference \div final\_value \times 100$ which $difference$ is the difference between the second file and the first file. In Table 3.1 are reported the data and the final overhead rate.

At the end, the overhead introduced by the idiomatic style is about the 4 percent which does not represent as huge problem.

3.4.2 Second experiment

As anticipated in Section 3.4, it will be considered two runtimes: the first one, called "Old Env" is the naive implementation without any extension/addiction, while the second one, called "New Env", is the new type of runtime in which it's implemented the migration mechanism. For the two phase, the instantiation and execution, it has been measured the time taken to complete those. To achieve this, it has been used the Instant structure of the standard library of Rust, which represents a monotonically non decreasing clock, for wrapping the instantiation and the execution section of the project. Finally, this procedure

has been repeated for thousand times. The Figures 3.2 and 3.3 depicts the obtain results.

Figure 3.2: Instantiation time of WASM module in two different environment

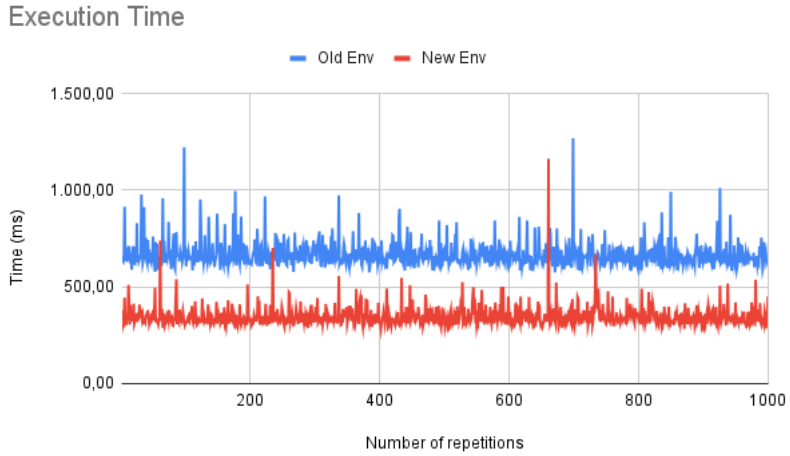

Figure 3.3: Restore time of WASM module in two different environment

The initial thing to observe is in the instantiation time, in Figure 3.2. The lines basically overlap each other meaning that the instantiation time hasn't been affected by the modification applied to obtain the new runtime. This is reasonable since the checkpoint and restore phases take place at runtime level, meanwhile the instantiation occurs before them. Therefore, the instantiation remains unchanged. The same thing cannot be said for the execution time time showed in Figure 3.3. There is a huge gap in the graph of the two runtime measurement, in order of two hundred millisecond. The main reason for this impact as been anticipated before, in the new runtime the checkpoint and restore

function take place at runtime, that is during the execution, thereby increasing the overall execution's time. At the end, as expected, only the execution phase is conditioned by the checkpoint/restore functions in the new runtime.

3.4.3 Third Experiment

Lastly, it will determine how the cost of checkpoint and restore phase changed during time. As in Section 3.4.2, the idea is the same but in this case it has been isolated the time taken for do the checkpoint and restore. A graphical visualization is presented in Figure 3.4 and 3.5. It is important to outline that a restore function is called at least one time in the lifetime of the WASM module, while the checkpoint function is called at least once. So, in the checkpoint graph, the single repetition represents the mean of all checkpoint's times that had been take place during the execution.

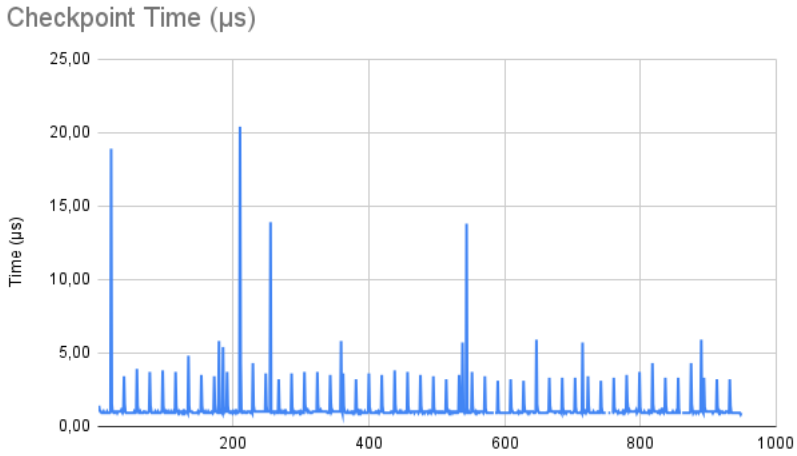

Figure 3.4: Checkpoint time of WASM module

What can be said is that in the restore graph, the time taken is basically constant expect for some outliers. Those values derives by the swapping of the memory during the execution, causing an delay in the copy of the content inside the linear memory. Meanwhile, in the checkpoint is present a sort of pattern which is repeated. This alternation creates two main groups and to better see this division, it has been applied the k-means algorithm. Basically, it's a algorithm that creates cluster of data which share common property. Also, it finds the values which represent the center of the cluster, i.e the value that has more similarities with all the members of the cluster. A detail and well explained description is presented by Steinley [40]. The algorithm has been run on the dataset of the times, asking to find three different cluster. It would be sufficient to use two, but to better visitation it has decided to use three since some outliers where affecting the subdivision. In the Figure 3.6, there are shown the two main of values.

The red dots represent one type of cluster, which is centred has value 0.977, while the blue ones represent the second cluster with center that has value 3.839.

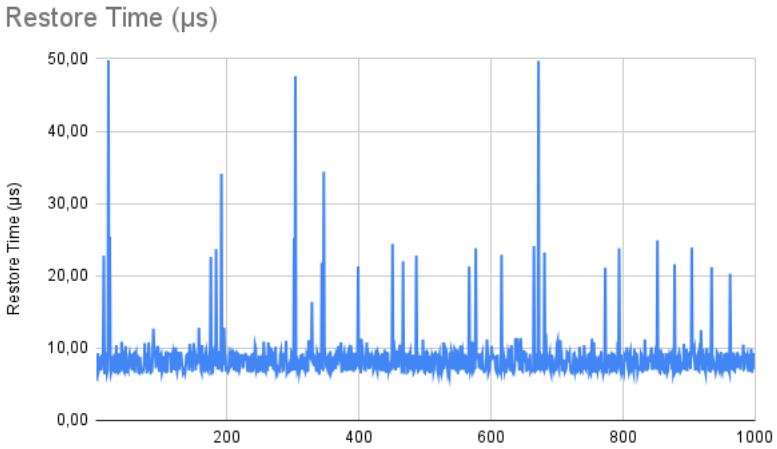

Figure 3.5: Restore time of WASM module

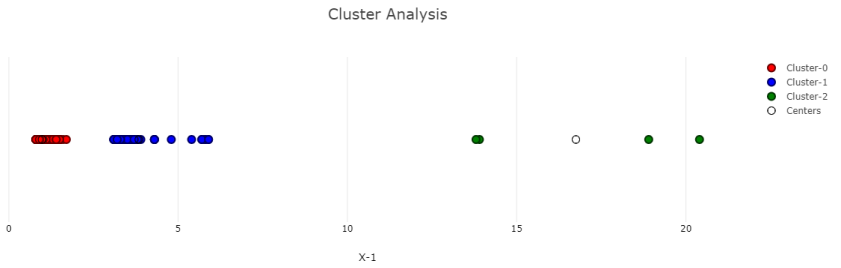

Figure 3.6: Result graph of k-means algorithm applied to checkpoint times

The green ones are the outliers which are not relevant. The nature of this data can be related by the locking phase inside the checkpoint function. In fact, every time there is mutex which it's created and then there is an acquired lock by the thread executing the checkpoint function. To demonstrate this, it has been measured the time implied to acquiring the lock in the mutex during the execution. However, the resulting data had a order too small to be compered, so instead it has been measure the checkpoint's time without the implementation of the lock. The Figure 3.7 shows the obtain results.

The times are around the value 1 in the graph, and in fact the average of all the measured time is 1 which is very similar to center of Cluster-1. So as supposed, removing the lock has decreased the time taken to perform the checkpoint. Thus, the overhead time is introduced by the locking system of Rust programming language. At the end, considering this example module the cost seems to be constant or, more specifically, not too variable. Although, as already pre-announced, more examples would lead to a better evaluation of this experiment.

A solution to the problem outlined in Section 2.1 has been presented. The technique of utilising checkpoints to save the state of the module and resuming

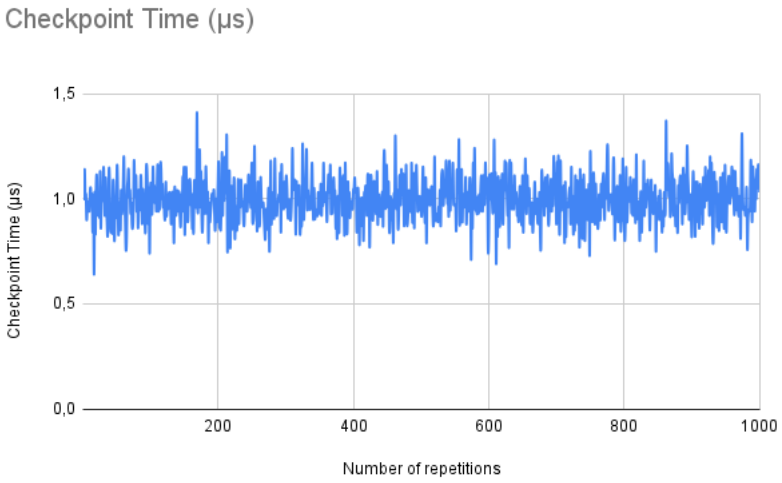

Figure 3.7: Graph of checkpoint times without the lock phase

it with restore has been illustrated and analysed in conjunction with its implementation. Subsequently, experiments pertaining to the "cost" of this solution are presented, and the results demonstrate that these "costs" do not detract from the solution's overall efficacy.

Chapter 4

Proceeds

This concluding chapter will demonstrate the outcomes of this work in relation to the objectives set out in Section 2.3. Subsequently, a discussion is presented regarding potential future work that could be conducted in relation to this project.

4.1 Key findings

Looking to the objectives described in section 2.3, it has been demonstrated that it's possible to implement a mechanism of migration based on a WebAssembly runtime. Moreover, since the heterogeneity of the devices it was still feasible to relay on a Just-in-Time compilation to obtain a proof of concept. As already announced in section 3.3, since there was no other solution to compare the result, this can be considered a little step further in the state of the art. Nevertheless, some crucial point can be taken out: first, the size of the generated artifact from the idiomatic programming style present a overhead which, due to is rate (about 4%), can be considered not relevant. Thus, from this point of view, the proposed solution doesn't worsen the state of art WebAssembly runtime. Second, the checkpoint operations were the more expensive in term of time consuming since every time there was a change in the state it must be committed, by creating and acquiring a mutex lock. Lastly, this kind of solution is not dependant on the underlying runtime, in fact it's possible to implement the functions checkpoint/restore in any runtime that can accept implementation on foreign functions. So it's possible to have different working runtime in the same distributed system.

Together with the advantages, some limitations are present to this project. First, force the developer to "rethink" is application in term of "array" and checkpoint/restore functions it can be not a good idea since the modality of how the runtime manages the states of the application has to be transparent to the developer. Second, the checkpoint/restore solution limits the range in which a module can be migrated, involving that if the request of migration happens before saving the new state, it must re-execute some operations to come back to that state. Ideally, the snapshot represents the state in which the module was at a precise operation, so that it's possible to resume the execution from the next one. Lastly, there is no kind of support for the re-connection of the modules

which use socket for expose and use web services. Since it was an object of this work, it's important to outline it.

4.2 What's next

This work represents an initial stage where continue to build up and lay the foundation for further works.

First of all, as already reported, the embedded environment compiles the WASM module. A possible alternative would be to use a technology that "interprets" the machine code from WebAssembly, in order to avoid other step of compilation. However, this solution will be request more execution time for module, since every instruction must be translate in native code, but it will be easier to determine whether the execution is stopped and restart form that point lately. Therefore, an investigation with benchmarks in the same filed of this work would be interesting.

Secondly, the web server uses the TCP/IP protocol to communicate over the network. This choice was made for sake of simplicity, in order to obtain an functional API web server. Recalling from section 2.3, one of the object was to investigate a solution to handle re-connection between hosts. The intuition was to integrate the QUIC protocol, which incorporate the ability to handle client re-connection, and use it on top of that an API web server. Moreover, this protocol utilize by default the TLS protocol at version 1.3 for encrypt the communications. So, exploring on this topic will improve not only the reliability of the system, but the degree of security of the entire system.

Thirdly, the previous point concerns only the client, but it's possible that in this system even an server host must be migrate. The QUIC protocol was not designed for this purpose, anyway Conforti[41] have proposed an extension of this protocol to handle re-connection of server host. Implementing this technology inside the system would achieve all the objective of this work.

Lastly, a WebAssembly component can have different resources in use in addition to connection. A simple example can be a file present in the host, or another WASM module which is a necessary library for the first one. In this scenarios, migrating an execution of a module would cause the its failure since there will be missing the necessary resources in the new host. Even this kind of situations must be handle in a system like this, so examine a possible solution would get close to a complete implementation of the system.

In conclusion, this paper has presented the Cloud-Edge Continuum paradigm, defining it and discussing its most significant applications. Nevertheless, the current state of cloud and edge computing does not permit its immediate utilisation, as its primary characteristic – the ability to migrate data between the cloud and the edge – is not yet fully realised. A potential candidate for unifying these two domains has been identified: WebAssembly, a novel low-level bytecode representation. It was therefore proposed that a possible migration mechanism for WebAssembly components over the Cloud-Edge Continuum be studied. Subsequently, the potential configuration of this prospective system was conceptualised, taking into account all the pertinent characteristics. The existing literature has not identified a solution to this problem, and thus the objectives of this study have been defined. Subsequently, a solution was devised and evaluated based on the findings of a series of experiments. In conclusion,

the results have been presented, along with suggestions for future work.

Chapter 5

Acknowledgements

This has been a long journey, in which there have been many insides and difficulties that have challenged me, especially from a mental point of view. Nevertheless, this is a circle that closes and the people who made this possible deserve to be thanked.

I thank Professor Tullio, who introduced me to the subject and supported me in the most complex moments by showing me the right way.

I thank my parents, who gave me the opportunity to embark on this experience.

I thank Cristina, who endured all of this at my side with love and patience, never burdening me with anything. You are still my rock.

Finally, I thank all the people I met during this journey, from the lecturers in the maths department to my fellow students who never forgot me. I know I hardly ever show it, but I love you all.

Bibliography

[1] D. Li, J. Shan, Z. Shao, X. Zhou, and Y. Yao, "Geomatics for smart cities - concept, key techniques, and applications," *Geo-spatial Information Science*, vol. 16, pp. 13–24, 03 2013.

[2] "Data volume of IoT connected devices worldwide 2019 and 2025," https://www.statista.com/statistics/1017863/worldwide-iot-connected-devices-data-size/, accessed: 2024-7-11.

[3] M. Taneja and A. Davy, "Resource aware placement of data analytics platform in fog computing," *Procedia Computer Science*, vol. 97, pp. 153–156, 12 2016.

[4] D. tecnologies. Edge to cloud continuum overview. [Online]. Available: https://infohub.delltechnologies.com/en-us/l/integrated-edge-management-in-smart-manufacturing-white-paper-1/edge-to-cloud-continuum-overview/

[5] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Webassembly as a common layer for the cloud-edge continuum," in *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, ser. FRAME '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 3–8. [Online]. Available: https://doi.org/10.1145/3526059.3533618

[6] U. F. Minhas, "A performance evaluation of database systems on virtual machines," 01 2007.

[7] H. Abdah, J. Barraca, and R. Aguiar, "Qos-aware service continuity in the virtualized edge," *IEEE Access*, vol. PP, pp. 1–1, 04 2019.

[8] D. Chernikov, D. Romanov, and S. Malyutkin, "Multiple inheritance and multiple dispatch in slang: the implementation approach for various target platforms," Ph.D. dissertation, 06 2020.

[9] The nist definition of cloud computing. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[10] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, p. 50–55, dec 2009. [Online]. Available: https://doi.org/10.1145/1496091.1496100

[11] Amazon. Amaon s3 storage. [Online]. Available: https://aws.amazon.com/s3/

[12] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1063–1075. [Online]. Available: https://doi.org/10.1145/3352460.3358296

[13] C.-H. Hong and B. Varghese, "Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms," *ACM Comput. Surv.*, vol. 52, no. 5, sep 2019. [Online]. Available: https://doi.org/10.1145/3326066

[14] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, "The internet of things, fog and cloud continuum: Integration and challenges," *Internet of Things*, vol. 3-4, pp. 134–155, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660518300635

[15] F. Ait Salaht, F. Desprez, and A. lèbre, "An overview of service placement problem in fog and edge computing," *ACM Computing Surveys*, vol. 53, 03 2020.

[16] The european cloud, edge and iot continuum initiative. [Online]. Available: https://eucloudedgeiot.eu/

[17] A. Gazis and E. Katsiri, "Middleware 101: What to know now and for the future," *Queue*, vol. 20, no. 1, p. 10–23, mar 2022. [Online]. Available: https://doi.org/10.1145/3526211

[18] What is a virtual machine. [Online]. Available: https://cloud.google.com/learn/what-is-a-virtual-machine

[19] Definition of visrtual machine. [Online]. Available: https://www.vmware.com/topics/glossary/content/virtual-machine.html

[20] A. S. A. Alhammadi and V. Siva, "Survey study of virtual machine migration techniques in cloud computing," *International Journal of Computer Applications*, vol. 177, pp. 975–8887, 11 2017.

[21] WebAssembly. Webassembly. [Online]. Available: https://webassembly.org/

[22] M. Developer. Javascript api for webassembly. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API#memory

[23] E. Aliyev, "Analysis performance of web assembly applications on cloud," in *1st INTERNATIONAL CONFERENCE ON THE 4th INDUSTRIAL REVOLUTION AND INFORMATION TECHNOLOGY*, vol. 1, no. 1, 2023, pp. 23–25.

[24] S. Sekigawa, C. Sasaki, and A. Tagami, "Web application-based webassembly container platform for extreme edge computing," in *GLOBECOM 2023 - 2023 IEEE Global Communications Conference*, 2023, pp. 3609–3614.

[25] P. Mendki, "Evaluating webassembly enabled serverless approach for edge computing," in *2020 IEEE Cloud Summit*, 2020, pp. 161–166.

[26] Principle of least privilege. [Online]. Available: https://en.wikipedia.org/wiki/Principle_of_least_privilege

[27] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," 2019. [Online]. Available: https://arxiv.org/abs/1902.03383

[28] T. Naumenko and A. Petrenko, "Analysis of problems of storage and processing of data in serverless technologies," *Technology audit and production reserves*, vol. 2, no. 2, p. 58, 2021.

[29] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 140–149.

[30] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 225–236. [Online]. Available: https://doi.org/10.1145/3302505.3310084

[31] W. Subgroup. Webassembly system interface. [Online]. Available: https://wasi.dev/

[32] M. Nieke, L. Almstedt, and R. Kapitza, "Edgedancer: Secure mobile webassembly services on the edge," in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 13–18. [Online]. Available: https://doi.org/10.1145/3434770.3459731

[33] B. Li, W. Dong, and Y. Gao, "Wiprog: A webassembly-based approach to integrated iot programming," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10.

[34] H.-J. Jeong, C. H. Shin, K. Y. Shin, H.-J. Lee, and S.-M. Moon, "Seamless offloading of web app computations from mobile device to edge clouds via html5 web worker migration," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 38–49. [Online]. Available: https://doi.org/10.1145/3357223.3362735

[35] Lunatic: a webassembly runtime inspired by erlang. [Online]. Available: https://lunatic.solutions/

[36] The erlang runtime system. [Online]. Available: https://blog.stenmans.org/theBeamBook/

[37] Checkpoint/restore in userspace. [Online]. Available: https://criu.org/

[38] B. Alliance. Wasmtime: a fast and secure runtime for webassemblya fast and secure runtime for webassembly. [Online]. Available: https://wasmtime.dev/

[39] Ergonomic and modular web framework built with tokio, tower, and hyper. [Online]. Available: https://github.com/tokio-rs/axum

[40] D. Steinley, "K-means clustering: a half-century synthesis," *British Journal of Mathematical and Statistical Psychology*, vol. 59, no. 1, pp. 1–34, 2006.

[41] L. Conforti, A. Virdis, C. Puliafito, and E. Mingozzi, "Extending the quic protocol to support live container migration at the edge," in *2021 IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2021, pp. 61–70.