EPFIOT: Edge provisioning for IoT

Sergio A. Semedi Barranco

MÁSTER EN INTERNET DE LAS COSAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Internet de las Cosas

Nota: 9

Julio de 2020

Directores:

Francisco D. Igual Peña
Luis Piñuel Moreno

Autorización de difusión

Sergio A. Semedi Barranco

2 de julio de 2020

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: Edge provisioning for IoT, realizado durante el curso académico 2019-2020 bajo la dirección de Francisco D. Igual Peña y con la colaboración externa de dirección de Luis Piñuel Moreno en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

El presente trabajo describe la implementación de EPFIOT (*Edge Provisioning For Internet of Things*), una *appliance* desarrollada para contemplar casos de Infraestructura como Servicio (IaaS, *Infrastructure as a service*) en el marco de *edge computing*, y específicamente diseñada y preparada para interactuar con dispositivos IoT usando aceleradores *hardware* de propósito específico en el ámbito del aprendizaje automático.

En un mundo gobernado por el paradigma *cloud computing*, surgen dudas sobre si es estrictamente necesario llevar a cabo un envío de todos los datos que recogen los dispositivos a Internet, afrontando la latencia que esto supone. EPFIOT persigue, a cambio de un uso contenido de recursos, ofrecer una infraestructura completa para llevar a cabo un procesamiento integral de datos obtenidos desde dispositivos en la propia red local, facilitando máquinas en el borde (*edge computing*) potencialmente equipadas con aceleradores de propósito específico para aprendizaje automático que realizan inferencia sobre los datos de forma rápida, reduciendo la latencia asociada a la transferencia a grandes centros de datos centralizados o distribuidos, pero en cualquier caso remotos.

La aplicación desarrollada ofrece una interfaz `GraphQL` que posibilita el desarrollo de un ecosistema IoT de infraestructura en el borde usando la virtualización de Linux (`kvm`) y otras tecnologías emergentes como LwM2M facilitando la configuración de los dispositivos.

Palabras clave

Internet de las cosas, Virtualización, Infraestructura como servicio, Computación en el borde, Computación en la nube, LWM2M, Red local, Aplicación, Enrutador, Maquina virtual.

Abstract

This document describes the implementation of EPFIOT (*Edge Provisioning For Internet of Things*), an appliance that follows an Infrastructure-as-a-service (IaaS) pattern targeting edge computing. EPFIOT is specifically designed and implemented to interact with IoT devices, and to use ad-hoc hardware accelerators for specific purposes in the field of machine learning.

Until the emergence of edge computing, cloud computing was the only choice for data processing derived from IoT. However, certain doubts arise about its strict necessity, and the requirement of sending all the data collected by the devices directly to Internet, paying for the (sometimes large) latency costs. EPFIOT aims, with a contained use of resources, to offer a complete infrastructure stack to perform an integral processing of data obtained from devices in the local network, simplifying the way of providing virtual machines at the edge, through the use of specific accelerators for machine learning that perform inference on the data quickly, hence reducing the latency associated with the transfer to remote datacenters.

The application exhibits a full `GraphQL` interface building an entire IoT ecosystem, using infrastructure on the edge thanks to Linux virtualization (`KVM`) and emerging technologies, such as LwM2M to provide device bootstrapping.

Keywords

Internet of things, Virtualization, Infrastructure as a Service, IaaS, Edge computing, cloud computing, LWM2M, LAN, local area network, appliance, router, Virtual Machine.

Contents

Chapter 1

Introduction

1.1 Motivation

Cloud computing has been a major word speaking about data compute in the last decade; as of today, cloud computing is used to store, clean, process and analyze virtually every chunk of information gathered from any source. With the emergence of Internet of Things (IoT), cloud computing is still a hot topic and an integral part in the deployment of complete infrastructures, providing easy and scalable access to applications, resources and services, and being fully managed by cloud service providers [1].

IoT devices provide large amount of data, and hence cloud processing is a natural solution to deal with them. The *Cloud of Things* (CoT) term comes here into play, deploying connected devices directly to the cloud to perform complex operations with the generated data using with Artificial Intelligence. This is the perfect tool for creating smart tasks that harness the immense amount of information. However, Cloud Computing faces several challenges to meet the most stringent performance requirements of many application services, specially in terms of latency and bandwidth [2].

To address and solve those problems, Edge Computing is a new paradigm that aims at bringing data storage and computation closer to the data source in order to improve response times and save data bandwidth. Edge Computing consists on increasing the resources available on the network edge, adopting a platform that provides intermediate layers for storing, cleaning, analyzing and inferencing. Regarding the layers located at the edge, it is obviously impossible to provide the same kind of datacenter capabilities that are currently available in Cloud Computing premises. Edge Computing implies limited computational capabilities, so one of the main problems to be solved is to get similar cloud platforms (at least in terms of capabilities, flexibility and deployment mechanisms) into an edge environment with certain compute power for processing data.

Specifically, an edge computing infrastructure is based on a set of local hardware resources, that we have also adopted in this work. Specifically, we base our development on a fairly common hardware infrastructure composed of two main elements, namely:

- **A Single Board Computer (SBC) or a minicomputer/barebone** that exhibit relatively high computing capacities with a proper reducing cost, energy consumption and form factor to adapt it to the specific scenario in which they are usually deployed.
- **Specific-purpose USB accelerators (DSAs)**, that provide co-processing capabilities to the central aforementioned computer (e.g. an Edge TPU as a Machine-Learning coprocessor). It accelerates specific tasks (e.g. inference processes for artificial intelligence models in the case of the Edge TPU), and can be selected depending on the typical application types that will be deployed on the edge infrastructure.

While the hardware problem is covered with the aforementioned elements, there are other elements involving the current paradigm of Cloud and Edge computing that are critical in order to develop an efficient and flexible edge architecture.

Virtualization is the capability of creating virtual machines that act like real ones. This technology becomes mandatory in Cloud and Edge Computing, since the creation of automatic elastic services requires getting rid of haphazard IT rooms, cables, and bulky hardware, reducing the overall IT overhead as well as management costs [3].

The creation and management of these virtual machines (VMs) would be impossible without the existence of **hypervisors**, pieces of software in charge of controlling the VM and providing a connection for the virtual resource and the real one.

Virtual appliances are VM images, usually with a specific configuration, designed to run on a virtual platform. They are designed to reduce or eliminate the installation, configuration, and maintenance costs associated with running suites of software [4].

When anything is able to connect to the Internet and generate data, there will be a possibility that at some stage uploading data to the cloud or sync device will not be necessary any longer. Temporarily, the data might not be required. In that scenario, either the device must be stopped from generating data or a gateway device must decide when the data upload should be discontinued to avoid using network and cloud resources for a while [5]. Having this in mind and knowing that in a near future millions of sensors (also known as things) will be connected to the Internet, the **IoT gateway** appears to serve as a bridge for connecting these sensors, send data to the cloud or participate with an edge approach. A way to implement this edge gateway might be building a virtual appliance that will be situated close to the things. Figure 1.1 shows a representation of the concept.

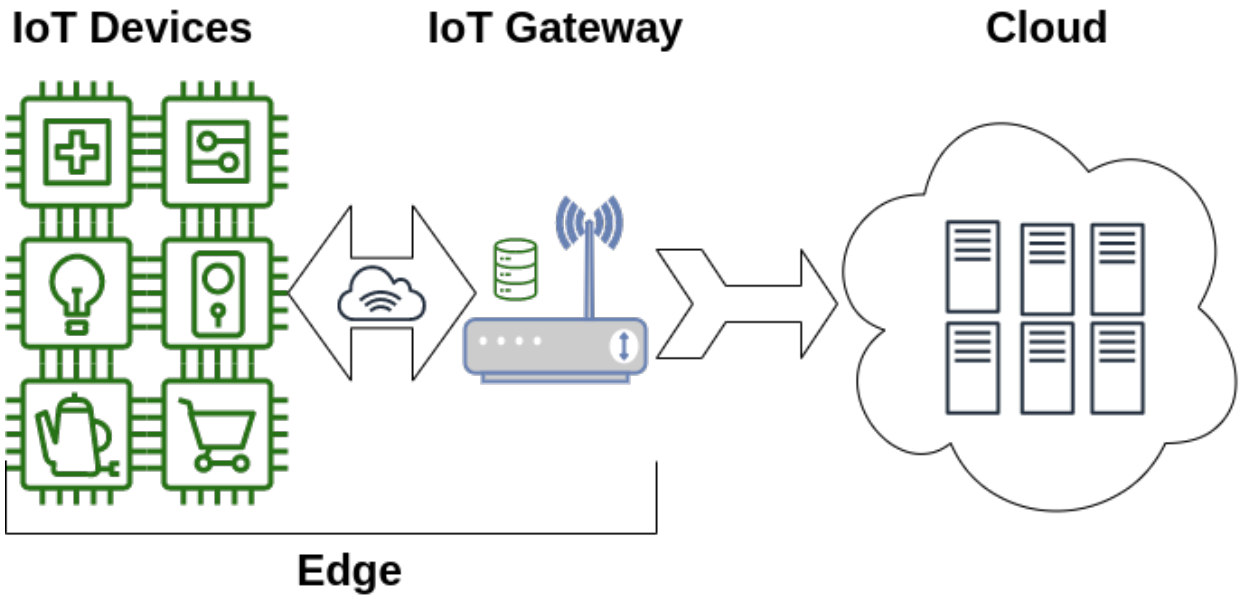

Figure 1.1: *Typical Gateway-based IoT deployment scheme.*

Some of the features that could supply the IoT gateway are the following:

- Communication bridge.
- Data storage, filter and aggregation.
- Device management and configuration.
- Security .
- Route data.

The presence of this Edge Layer facilitates the implementation of collaborative computing across devices, and it helps to apply data management policies. However, this edge layer is not usually rich in computing resources (mainly due to energy consumption limitations), which dramatically limits the approaches and designs that could be implemented to port move computing tasks to the edge. Following the Cloud-oriented approach of hypervisor-based virtualization, it can be proven that it can be cumbersome on these devices [6]. However we will see that this type of approach can be of great help using the proper hypervisor in order to gain the maximum efficiency.

1.2 Objectives

The main goal of the project is to prove that an IoT gateway using hypervisor-based virtualization can be completely functional and can satisfy all the requirements imposed by edge computing, including resource management in terms of USB accelerator for neural network inference.

Therefore, with the purpose of showing some actual results, a virtual appliance (EPFIOT) has been created running a custom service developed entirely from scratch using novel technologies for virtualization and communication.

With our appliance working, the next features will be evaluated on a real edge-level machine:

- Inference with virtual machines using real USB accelerators.
- The feature of share real USB accelerators between multiple virtual machines.
- Dynamic resource allocation including CPU and/or memory.

1.2.1 Use case

Under the principles previously established it has been defined a use case that allows the appliance development to have a concise scenario in order to achieve better results. The project is aimed at apartments blocks, takes into account a modern building in which every house can be considered an smart house, having a large quantity of sensors across the area.

Consider an external provider or a neighbourhood community that wants to take advantage of EPFIOT. That provider will be referred as the *client* in the following:

- A low cost board will be deployed in base building of flats/apartments as an IoT gateway with the EPFIOT appliance installed.
- The client will provide each apartment with sensors (temperature, humidity, camera, gas, ...).
- The EPFIOT appliance will have specific-purpose (neural network) accelerators physically connected through USB ports.
- The EPFIOT appliance will provide a service to the client allowing the spawn of customized VMs that have direct access to the accelerators.

- The client will be able to configure sensors using EPFIOT.
- The client will have full access and management for the VMs and he/she will be able to allocate some resources including the USB accelerators.
- Sensors will generate data, instead of sending this data directly to internet, a proper virtual machine could receive the data and use a accelerator to perform inference.
- Results could be stored in EPFIOT or another application in a low latency context.

Figure 1.2 shows a representation of the case listed above.

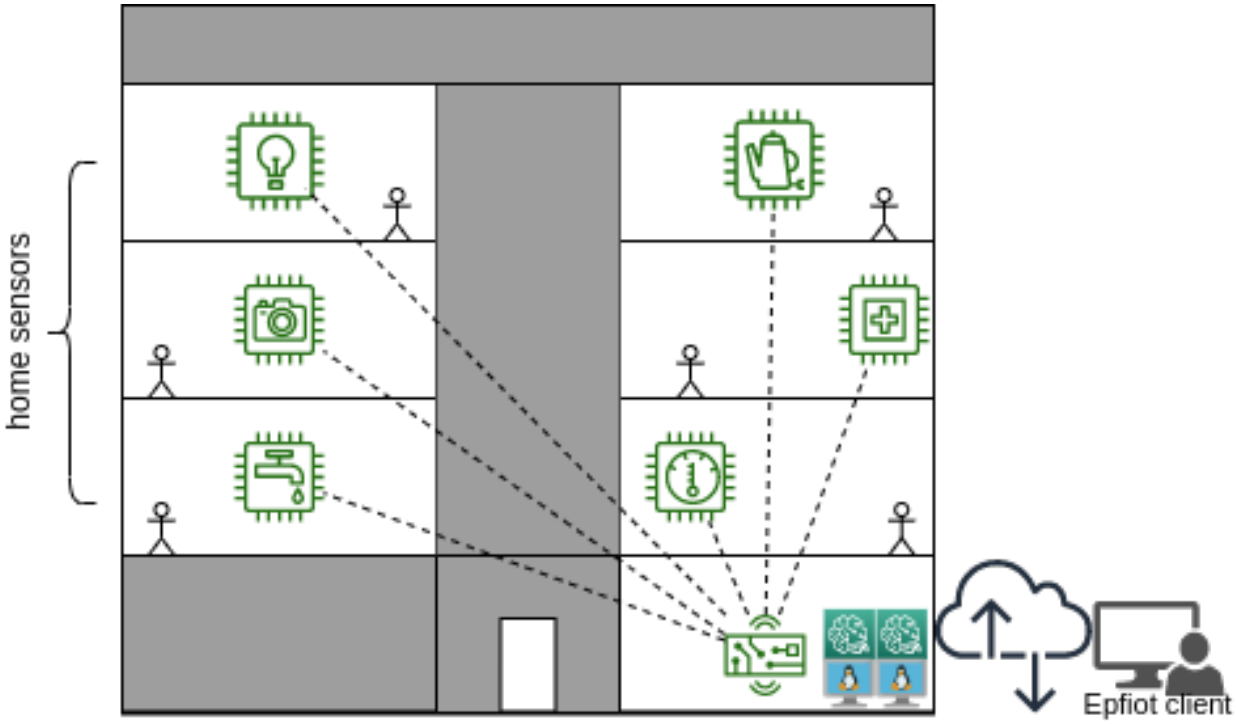

Figure 1.2: EPFIOT *use case example.*

1.3 Document Organization

This document contains 7 chapters described as follow:

- **Chapter 1, Introduction:** This chapter introduces a set of conceptual definition that will be mentioned along the document. This chapter also displays the complete project 's purpose through the explanation of its motivations and goals.

- **Chapter 2, State of the Art:** This section displays the current state of Edge computing from a infrastructure point of view, discussing the underlying technologies together with the advantages and disadvantages of each option. Finishing by a brief overview of related projects.
- **Chapter 3, Development Environment:** A list of the main technologies used for Epfiot will be enumerated, underlining what kind of utility is used for the project. Hardware and requirements are also discussed in this chapter.
- **Chapter 4: Architecture**: The chapter describes the overall architecture of Epfiot, reviewing every component and how they are connected between each other.
- **Chapter 5: API:** This part explains in detail the functionality of the project, end-user's operations are listed in this chapter.
- **Chapter 6: Examples:** This section is about showing the final results that are obtained with the Epfiot application running. In order to do so, certain examples will be presented in which a completed use of the application is required.
- **Chapter 7: Conclusions and Future Work:** Finally a brief summary of the project commenting what kind of implications it has and how to improve the current status of the project in a near future.

Chapter 2

State of the Art

This chapter describes the current state of Edge computing related to Internet of Things. Different types of architectures that diverge in core technologies regarding edge environments will be discussed; in addition, we will also cover some projects that provide *similar* solutions as EPFIOT. Chapter 1 has covered some basic definitions used in the project such as *virtualization*; however, this is not the only option regarding technologies that allows to prepare the proper infrastructure in an edge context.

2.1 Containers: a lightweight alternative

Virtualization has been around for a long time but since a few years ago, containers have been adopted globally, transforming the IT world as we currently know it. A **Container** is a set of one or more processes that are isolated from the rest of the operating system. These process would lately be very useful to prepare independent components that share resources with the host machine and have portability. Containers are a form of virtualization, while virtualization works at hardware level emulating all the resources that are needed for a guest operating system to run. Containers, in the other hand, work at the operating system level being a lightweight alternative.

The idea of the container technology first appeared in 2000 as FreeBSD jail, a safe environment that a system administrator could share with multiple users. Later on, more technologies were combined to become this isolated approach a reality. For example, *control groups* –`cgroups`– is a kernel feature that monitors and limits resource usage for a process or groups of processes or tt systemd, an initialization system that sets up the userspace and manages their processes. In 2008 Docker was born with a new container technology adding new concepts and tools, layered images, command line tools, a server daemon granting,

granting to the user the ability of build new containers quickly and share between others [7].

2.2 Virtualization and Containers, what to choose

The proper selection of an underlying technology to manage edge infrastructure takes into account several factors, namely:

- System software that will run at the top of the architecture fast and lightweight. Designed to operate directly on edge devices (computation, network, storage) and for this purpose, normally system software needs the support of multi-tenancy and isolation.
- Resource constrained because edge devices equip smaller processors and limited power budget 1.1. Managing these resources properly is one of the challenge that edge computing needs to confront.
- Elastic escalation, depending on the services that are going to be provided.
- Network capabilities.

Usually, in an edge computing scenario, several applications are running from different tenants. Due to the factors previously discussed, both containers and virtual machines are valid approaches in order to satisfy edge requirements [8]. Both enable fault and performance isolation between tenants and can limit the accounts for the resource usage.

However there are differences, see Figure 2.2:

- Virtual Machines are built including virtualized resources that emulate the entire computer such as CPU, memory, network, storage, GPU, ... This means that the tenant installs the operating system and runs applications like in a real physical machine. Virtualization isolates completely the execution environment at the exchange of (usually non-negligible) performance penalties.
- Containers are lightweight virtualization and are multiplexed by a single Linux kernel. They do not require an additional layer if you compare them with the virtual machines. They share the same OS kernel, keeping the isolation principles by using `cgroups` and Linux namespaces. This allows the containers to achieve performance similar to the native environment.

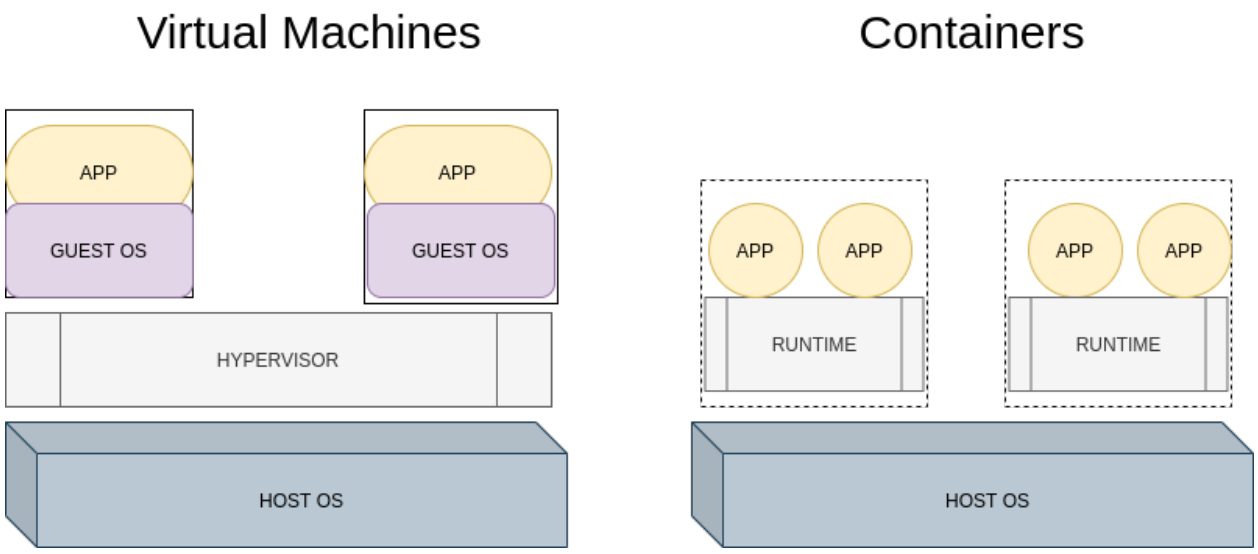

Figure 2.1: *virtual machine architecture and container architecture*

Virtual Machines	**Containers**
Heavyweight	Lightweight
Each VM runs in its own OS	All containers share the host OS
Hardware-level virtualization	OS virtualization
Startup time in minutes	Startup time in milliseconds
Allocates required memory	Requires less memory space
Fully isolated and hence more secure	Process-level isolation, possibly less secure

Table 2.1: VMS and containers comparison [10].

Choosing a technology that best fits in a edge environment is not an easy question. Virtualization approaches vary from system to system. Many recent works chose lightweight OS-Level virtualization like LXC to run services on edge systems that have limited storage and processing capabilities. Unlike virtual machines, multiple containers share the same Linux kernel and therefore exhibit very small overhead [9].

Taking into account the lightweight capability of the containers reducing resource consumption it could be said that is the perfect solution for Edge computing. However it may face security issues that are solved in a virtual machine architecture with better isolation of the environment.

In order to compare both technologies, a table has been provided, see Table 5.7.

However, the EPFIOT project takes into account the drawbacks of virtualization, and adds new features of virtualization to improve the solution, namely:

- Baremetal performance using Linux hypervisor.
- Hardware passthrough.
- Lightweight VM images.

These features balance both technologies; on an edge approach it would be better to use containers if the service requires a high level of escalation on demand and the security issues are covered. Moreover in a scenario that this high elastic escalation is not required and some third party hardware is being used (e.g. USB accelerators), a virtualization approach would be a better fit.

2.3 Related solutions for edge computing

Different projects focusing on the same problem have been proposed using different perspectives. Some of them are listed next.

2.3.1 Openstack for *cloudlet* deployments

Openstack [11] is a cloud operating system that controls large pools of compute, storage and networking resource throughout a datacenter, managed and provisioned through APIs with an authentication mechanism. A dashboard is also available, giving administrators control while empowering their users to provision resources through a web interface.

Cloudlet [12] is a concept that emerges from the convergence of cloud computing and mobile computing; it is a mobility-enhanced small-scale cloud datacenter located at the edge of the internet. Its main purpose is supporting resource-intensive and interactive mobile applications providing powerful computing resources to mobile devices with lower latency.

Openstack was built thinking in large scale IT enterprises; its relevance is mainly associated with its open-source nature and the massive size of its supporting community. OpenStack provides flexibility and adaptability, so that the programmer can create distributions following his ow needs.

Openstack++ [13] is a distribution that extends OpenStack to leverage its open ecosystem bringing the cloudlet concept into OpenStack, while changing the VM model with the following features:

- *Import Base VM*: Import a Base VM from a image storage.

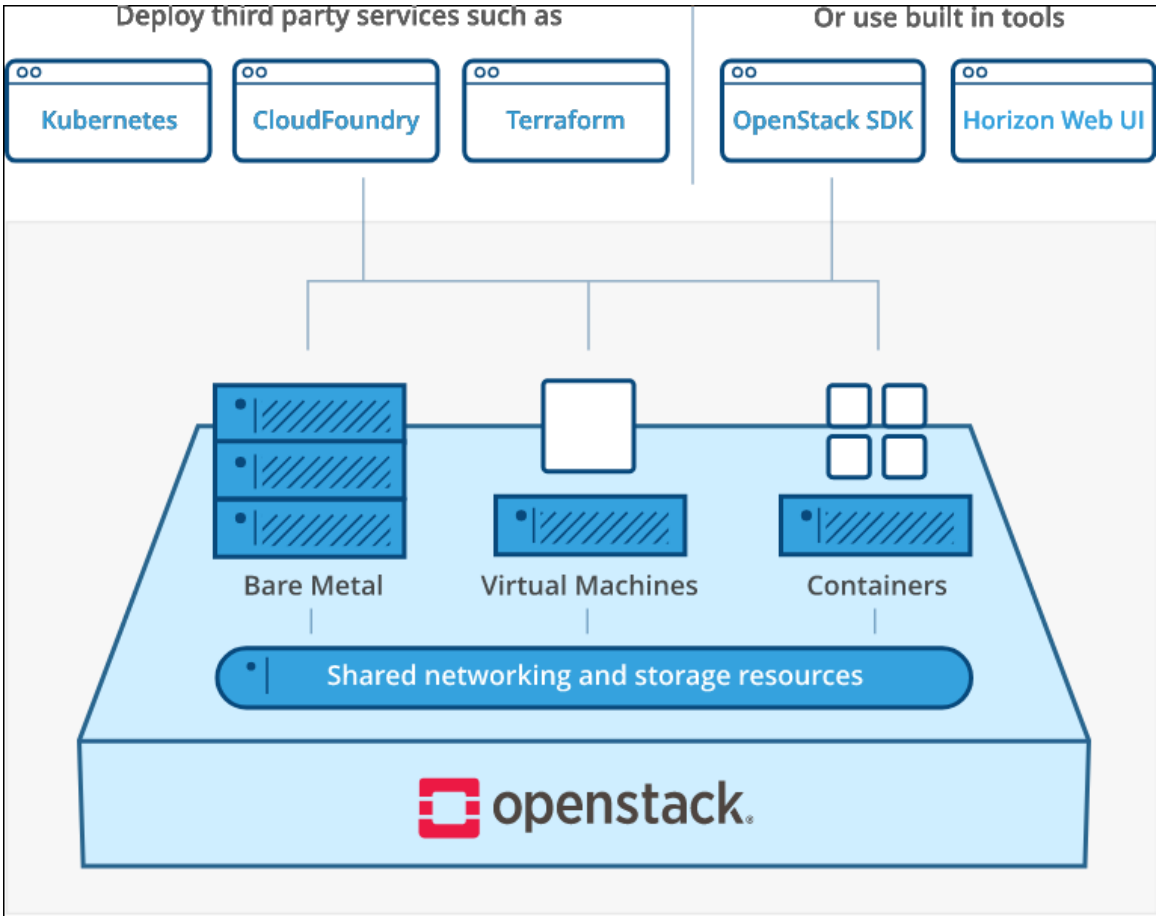

Figure 2.2: *Openstack overview, from Openstack official page.*

- *Resume Base VM*: Resume one of the base VMs to make a customized VM and to create a new VM.
- *Create VM overlay*: Create a VM overlay from a running VM instance.
- *VM synthesis*: Provisioning a VM instance at a OpenStack++ cluster using a VM overlay.
- *VM handoff*: Migrating a VM instance to a different OpenStack++ cluster.

This distribution was implemented by researchers at Carnegie Mellon University.

2.3.2 OpenNebula and Telefonica Onlife

OpenNebula is a cloud computing platform that features similar features than those provided by Openstack, but relying on simplicity over a complex plugin architecture.

Telefonica is utilizing OpenNebula to prototype a new generation of Central offices that allows the deployment of programmable services rather than the traditional black-box solutions. Telefonica aims at virtualizing the access network and give third-party Internet of Things application developers and content providers cloud-computing capabilities at the network edge [14] through its Onlife project.

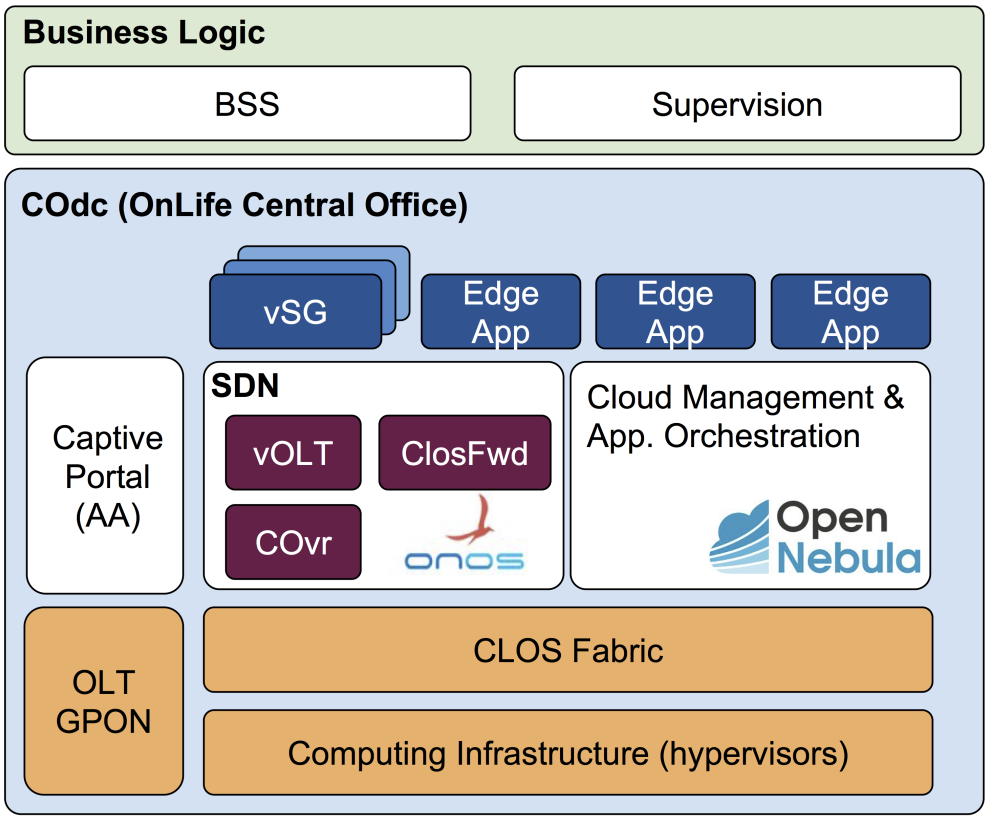

Figure 2.3: *Telefonica Onlife project.*

OneEdge

OneEdge is an OpenNebula-based distribution that fits into edge computing. This distributed cloud management platform aggregates on-demand cloud resources across multiple edge locations to enable innovate and low-latency next-generation services [15].

The main principles of OneEdge are:

- Backward compatibility with existing VM & container appliances.
- Leverage upcoming ecosystem of hyperscale and edge clouds.
- Easily combine centralized cloud resources with edge resources.
- Reuse proven cloud management capabilities.
- An open-source solution with a growing open ecosystem.

All these features make OneEdge an interesting option when installing an open source platform int he field of edge computing.

2.3.3 AWS

Amazon Web Services is the main company (product) speaking about cloud services nowadays. Basically, Amazon is the principal election when it comes to the public cloud. This large-sized company features a huge amount of products/services and they even have an IoT/Edge computing section.

AWS Outposts [16] is a fully-managed service that extends the whole AWS ecosystem; this new service provided by Amazon can be set forth in a on-premises facility and combine it with public cloud to generate a hybrid experience.

AWS Outpost is ideal for workloads that require a low latency access and local data processing. In other words: it is suitable for edge computing.

Chapter 3

Development environment

This chapter explains the technologies and hardware components used in the project. EPFIOT is delivered in the form of an *appliance*, that is, a proper virtual image that has been prepared specifically for the project; this *custom* image will run software developed for EPFIOT in particular (in fact, it has the same name) from scratch.

Details of this software are found in this chapter because it is the most complex part. However there are many factors that are needed to take into account in order to get the complete overview of the project. For example, real host preparation, the appliance content itself and the already tuned image prepared directly for the client's use. Next sections contain a summary of the whole environment. The project can be found in Github (https://github.com/Semedi/epfiot), and it is available under a MIT license.

3.1 The host

The Host is the physical machine that will support and run the project, so it is considered a critical and important part of it. EPFIOT is designed to run in an edge context, that means that the machine should cover some basics as described in chapter 1.1 and chapter 2.1. However EPFIOT has some additional minimum requirements that the host needs to meet.

3.1.1 Minimum requirements

The minimum requiresments, from the hardware and software perspective, dictated by *Epfiot* for a proper use are:

- Linux Support kernel $>$ 2.6.2.
- 64 bits architecture.
- Intel Virtualization Technology (VT-x) or AMD's equivalent.
- Intel Virtualization Technology for Directed I/O (VT-d) or AMD's equivalent.
- Compatible USB accelerator (in case it is needed).

Those requirements should be accomplished in order to reach the desired virtualization for the project. Intel Virtualization Technology [17] enables a VM to run at near-native speed. USB accelerator support will be discussed in Section 3.1.4

3.1.2 Software Requirements

The EPFIOT appliance can run in every virtualization platform; however, the provided project driver is prepared to work under the following premises:

- KVM kernel module as the hypervisor.
- QEMU as the hardware emulator.
- `libvirt` as the virtualization interface.
- OpenSSH server, for custom operations requested by the appliance.

KVM [18] stands for Kernel-based Virtual Machine and is a full virtualization solution for Linux that can work with some virtualization extension (as mentioned in Section 3.1) such as Intel VT-x. It consists of a loadable kernel module that is the core virtualization infrastructure, bringing to the VMs the possibility of interacting with the host real hardware.

Basically, KVM turns the Linux kernel into an hypervisor taking into account some features like memory management or process scheduling. KVM has a userspace API and it is exposed via `ioctls`. **QEMU** will use this userspace API. This software is the responsible of creating the hardware with a guest operating system running on top. QEMU [19] interfaces with Linux, especially the KVM module within it, to directly run virtual machines on the physical hardware.

libvirt is an open source API, daemon and a set of tools for managing this virtualization environment [20].

Finally an OpenSSH server is needed to perform some basic IO operations between the host and the guest (EPFIOT). The server should be configured only allow passwordless operations with a user capable of managing the guest from KVM.

3.1.3 Networking setup

There are different options in order to configure the host networking part; first, it is possible to create an ad-hoc Linux bridge manually. However, it is advisable to use the networking mechanism that the `libvirt` infrastructure provides through its virtualization API. In order to define a Linux bridge (alternatively referred as a Virtual Switch), a new virtual device in the host server needs to be created, so that virtual machines can leverage it as a 'plug in', redirecting network traffic through it.

In addition, by using the `libvirt` infrastructure, additional security concerns are considered out-of-the-box, reducing the impact on the developer side.

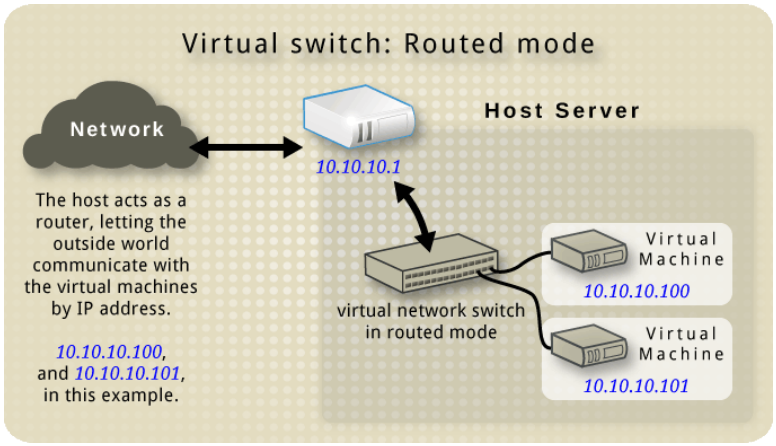

Figure 3.1: *Libvirt routed network [21].*

With *routed* mode, the bridge is connected to the physical host LAN, passing the guest network traffic back and forth without using NAT. This bridge sees the IP in each packet and decides necessary actions. This mode gets all virtual machines in the created subnet, routed through the bridge. The downside is that it is necessary to configure routers in order to get full access to the physical network, see Figure 3.1.

In addition, EPFIOT takes benefit from `libvirt` managed network in another way. By leveraging its DHCP feature, it is possible to generate virtual machine IP addresses on demane, making the effort of networking easy through dynamic (virtual) network cards.

At this point, kernel parameters such as IP forwarding, and the proper NAT rules should be configured. The EPFIOT project comes with a set of networking scripts that makes this

task easier.

3.1.4 Tested Hardware

Board

In the execution of the project we have used an Intel NUC NUC5i5RYK with all the aforementioned requirements. The Intel NUC is an appealing compute box for edge computing, mixing low power consumption with a relatively high performance and expansion capabilities. The following table provides detailed specifications for the device:

Intel NUC5i5RYB	
Internal drive	M.2 SSD
Processor	Intel® Core™ i5-5250U
Cores	2
Threads	4
Base Frequency	1.60 GHz
Turbo Frequency	2.70 GHz
Memory size	16 GB
Memory Type	DDR3L-1333/1600 1.35V SO-DIMM
Memory Bandwidth	25.6 GB/s
Memory Channels	2
Max. DIMMS	2
Graphics	Intel® HD Graphics 6000
USB ports	6
VT-x	Yes
VT-d	Yes

Table 3.1: Intel NUC specifications [22].

USB accelerators

As explained in Section 1.1, EPFIOT provides the ability of using virtual machines connected to real USB devices, and specifically, USB accelerators specifically designed for neural network inference, taking advantage of the compute performance when performing inference processes. This kind of process is actual of paramount interest in Edge Computing.

A clear example is the new Google Coral USB accelerator, used as an illustrative example in this project. The Coral accelerator is a complete toolkit (hardware and software) to build

products with local AI, specifically focusing on neural network inference. The on-board Edge TPU co-processor is capable of performing 4 trillion operations per second (TOPS), using 0.5 watts for each TOP. In the following table, we provide detailed specifications for the device:

Google Coral	
ML accelerator	Google Edge TPU coprocessor
Connector	USB 3.0 Type-C
Dimensions	65 mm x 30 mm

Table 3.2: Google Coral specifications [23].

Notice that there are several USB accelerators that could work with the EPFIOT solution in a similar manner to that tested for the Google Coral accelerator; we have chosen this specific device as a mere example and due to its popularity and ease of use.

Figure 3.2: *Devices used in* EPFIOT.

3.2 The Appliance

The nuclear part of the EPFIOT project is the appliance. A virtual machine has been prepared for the sake of serving a custom application that has been designed, developed and implemented thinking in this Edge Computing particular scenario.

The EPFIOT appliance is responsible of managing virtual machines, altogether with the devices connected to the host (e.g. USB accelerators), providing a multi-tenant environment, storing a persistent model and bringing a secure interface to the user among other things.

In the next sections, we provide further details regarding the selected technologies used in the development of the appliance. The architecture of the application will be discussed in the next chapter.

3.2.1 Operating System

Alpine Linux is the selected appliance for the guest Operating System. It is a security-oriented, lightweight Linux distribution based on MUSL `libc` and `busybox`. The decision was made taking into account the small footprint that the image leaves after installing Alpine and the resulting low consumption in terms of computing resources.

The image is stored in **qcow2** format; `qcow` stands for *Copy on Write*, a very common term in the computer science field and uses a disk storage optimization strategy that delays allocation of storage until it is actually needed, that is, after a write. Basically, the image will grow only if the mounted file system needs more space. This is the format used not only for the application, but also for all virtual machines that `Epfiot` would manage.

3.2.2 The Epfiot Package

Once the host part is covered and a virtual machine created with the operating system selected, it is time to explain the most complex part of the project: the Epfiot software, a custom package that has been developed exclusively and from scratch as the nuclear part of the project.

A brief scheme of the Epfiot appliance within the overall architecture of the project covered in this chapter is shown in Figure 3.3.

The Epfiot package is a service that runs in the appliance, composed by two major components:

- Epfiot main application: https://github.com/Semedi/epfiot
- Epfiot *bootstrap service*: https://github.com/Semedi/epfiot_bootstrap

We explain next the underlying technologies used in both components. On the other side, this will be used as an introduction for Chapter 4.

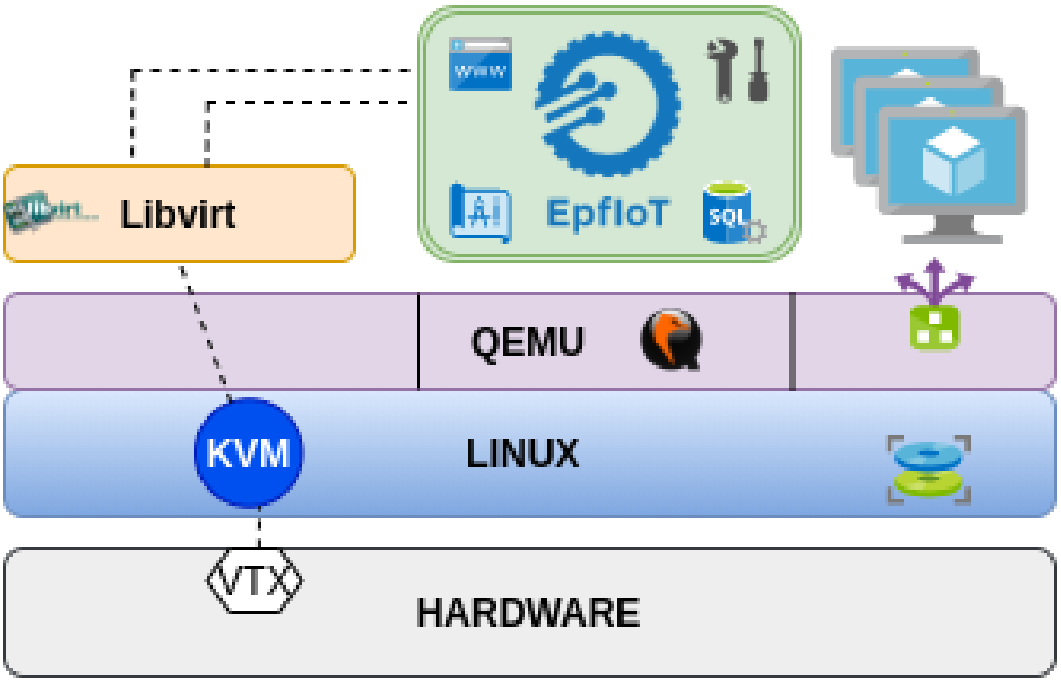

Figure 3.3: EPFIOT *appliance overview.*

The EPFIOT main application

The EPFIOT main application is the heart of the project; it is a piece of software built from scratch that aims to fill all the purposes related in Chapter 1.3. Before performing a deep dig into the architecture and the API interface of the project, the main technologies related to it are arranged in order to grant a clear view to the reader.

Dependencies

1. `alpine-sdk`
2. `libvirt-dev`
3. `libvirt-qemu`
4. `cdrkit`
5. `golang` > 1.10.4
   - 5.1. Epfiot
   - 5.2. libvirt-go
   - 5.3. graphql-go

5.4. sessions

5.5. logrus

5.6. gorm

5.7. yamlv2

5.8. sha512_crypt

The complete application has been developed using **git** as a version control system and it is hosted on Github. **Golang** was the selected language for this application. Go is an open source programming language that makes it easy to build simple, reliable, and efficient software. Some of the advantages of using Golang that benefits EPFIOT are:

- Language Design: unlike other languages like C++, the designers of the language made a conscious decision to keep the language simple and easy to understand.
- Binaries: allowing the maintainer to administer a compiled version of the software. In this case the whole appliance is provided but it is possible to download the standalone EPFIOT distribution and install it on any x86 platform.
- Powerful standard library: As a system language this feature allows the developer to perform system calls and I/O operations in a safer way.
- Static Typing: Saving time debugging run-time errors.
- Concurrency Support: Ideal for a multi-tenant Edge Computing service.
- Package Management: Making easy to deal with dependencies.

Knowing the foundational stones of the project, we show next some of the key technologies used in the project:

Libvirt, Qemu, KVM: The host operating system must feature the `libvirt` daemon running in the background (it is recommended to have the service enabled on `systemd`). On the appliance side, `libvirt-qemu` and `libvirt-dev` are needed. `libvirt-qemu` is a client of the `libvirt` daemon; on the other hand `libvirt-dev` contains the proper development headers. `libvirt-go` which is a golang library that acts as a binding for those headers, allowing the developer to build `libvirt-tools` using golang.

`libvirt` uses XML for template definitions, which means that every resource created with `libvirt` must exist in a XML form. Managing templates with XML is a tedious thing; that is why `libvirt-go-xml` is brought into scope. This library handles the complex XML operations for building the document leaving behind a much simpler interface. EPFIOT

makes heavy use of this library for building the templates. Those templates are sent to the host daemon using the `libvirt` client. Finally, the `libvirt` daemon in the host side, knows how to instantiate the templates emulating the hardware on QEMU and granting the KVM hypervisor capabilities for managing the guest.

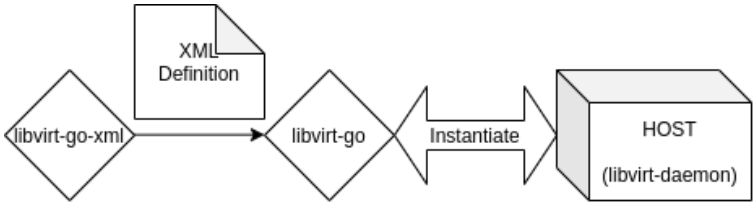

Figure 3.4: *Libvirt basic scheme.*

`libvirt` [24] includes a basic protocol and code to implement an extensible, secure client/server RPC service. That is why EPFIOT is able to communicate with the host in order to perform VM managing operations. Trying to get a secure environment, EPFIOT uses this RPC protocol tunneled over SSH. EPFIOT benefits from this situation using SSH for other mandatory tasks such as file system operations on the host machine.

After virtualization basics have been covered, let us move to the network management part. **Net-http** and **graphql-go** correspond to EPFIOT's service module. This module will be covered in detail all together with the other components in the next chapter; however, the technologies regarding this module are explained next.

The Golang language provides a convenient library to create `http` servers. EPFIOT uses this library in order to build an interface to the end-user. At the time of building this interface, the most common design is usually a REST API; nevertheless, a new technology was discovered replacing REST. **GraphQL** [25] is a framework with a query language that is used to express the data retrieval request issued to Web servers. This query language will behave taking into account an schema that the developer should define.

`Graphql` gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools. These characteristics make `graphql` very suitable for IoT; building this kind of service will save a lot of time since it does not require the design of new endpoints for every operation that the creator needs to expose.

`Graphql-go` is a `graphql` implementation written in Go; EPFIOT uses this library to build a service module that serves as an `http` interface for user interaction. `Epfiot` provides a schema for `graphql` that defines the end-user interaction with the system. More details about this schema can be found in Chapter 5.

The final point of interest is the persistence model. **Gorm** is a golang library to work over this situation. It provides some interesting features such as a complete ORM model,

Associations, Hooks, Transactions, SQL builder, etc. EPFIOT uses `Gorm` in order to build a persistent model that a multi-tenant virtualization service should meet. This model is stored in a SQL database. For development purposes, **sqlite3** is the desired option, but for production environments is recommended to use MariaDB or PostgreSQL.

The EPFIOT bootstrap service

In order to bring some interesting IoT-related features into EPFIOT, another piece of software was developed. The EPFIOT bootstrap service is a project that brings the **Lightweight M2M** protocol into this edge ecosystem.

LWM2M [26] is a Open Mobile Alliance standard that provides device management and service enablement capabilities for managing IoT devices. This protocol provides a light and compact secure communication interface along with an efficient data model. It is composed by a LWM2M Client (M2M device) and a LWM2M Server (M2M platform), a client-server architecture on top of COAP using UDP/SMS as transport bindings. The architecture main's target are constrained devices that require an efficient bandwidth usage [27].

Some features provided by LWM2M are the following:

- *Bootstrapping*: LWM2M Bootstrap Server is able to manage keying, access control and configuration of a device and enrol it with a LWM2M Server.
- *Device Discovery and Registration*: the device (LWM2M client) permits the server (LWM2M server) to know its existence.
- *Device management and service enablement*: allows LWM2M server to perform device management, sending operations to the client.
- *Information reporting*: Allows L2M2M to report information to the server.

The EPFIOT bootstrap service, as its name suggest, is a bootstrap server working with LWM2M that will handle the device bootstrapping phase, allowing those devices to register with standard LWM2M.

The decision of creating a different project instead of performing a direct integration in the main package was made considering that L2M2M is a young protocol and find a valid implementation turned into a very difficult task.

Exploring the possibilities for the project's implementation, we decided to use **Wakaama**, an implementation made by Eclipse Foundation of the Open Mobile Alliance's LighWeight M2M protocol written in C, designed to be portable on POSIX compliant systems.

The whole project can be found at: https://github.com/eclipse/wakaama, this project provides the LWM2M for the C language and a few examples about LWM2M components such as a bootstrap server, server and client. EPFIOT has forked this implementation to build a custom bootstrap server that will work along the EPFIOT main package to provide those capabilities to the project. This package will listen to Epfiot requests in order to register devices and machines imported directly from the ORM model previously discussed.

3.3 Tuned image

The last section of this chapter introduces another image that is not the principal appliance. This tuned image is built specifically for the EPFIOT project and is the base image that the virtual machines spawned by EPFIOT will use. It is the only choice at the moment: any client using EPFIOT is free to use whatever OS image it prefers. However, if the clients needs to take advantage of all EPFIOT features, the main recommendation is to use the prepared image; otherwise certaing components will need to be installed manually. The components of the base image are:

- Ubuntu 18.04 as operating system.
- Cloud-init for instance initialization.
- LWM2M server enabled over Websocket.
- TPU runtime for Google Coral accelerator.
- Example projects.

Ubuntu is the elected operating system due to it is the Linux distribution that supports a wide variety of environments, for example the TPU runtime package needed for the Google Coral accelerator works perfectly with Ubuntu and 18.04 is a stable version.

Cloud-init is a service installed inside the image that allows EPFIOT to provide VM configuration bootstrapping. When a virtual machine is created, EPFIOT asks the user for a configuration, like for example a username and a password. EPFIOT creates a CDROM owned by the virtual machine, the cloud-init service will read this CD-ROM at boot-time executing the configuration defined by the user. Cloud-init was created by Canonical and it is adopted in many companies and institutions such as AWS with the same purpose as that of EPFIOT.

LWM2M server was already explained in Section 3.2.2. This binary is enabled as a service at VM boot time and it is provided to the user via a Websocket. Some adjustments

are still required, but this grants the possibility of having a platform that is able to register devices in an automatic way.

With the components previously discussed, the Google Coral TPU runtime is installed with the image, allowing to perform inference operations using real hardware. Remember that the machine is already prepared to have a device and KVM is the responsible of perform the passthrough operation so the client experiences the use of a normal machine with a connected device handled by `udev`.

How this service is able to communicate with the main EPFIOT package will be explained, together with the application's architecture in the next chapter.

Chapter 4

The EPFIOT Architecture

In this chapter, we discuss more advanced aspects of EPFIOT architecture. To carry out this project, the EPFIOT main application has been written from scratch, except for the bootstrapping service part, based on top of the Wakaama implementation as already seen in Chapter 3.2.2.

EPFIOT consists of two separate pieces of software, namely:

- **Epfiot Go platform**: Golang Service with the main application.
- **Epfiot bootstrap**: C/UDP Socket with the LWM2M logic.

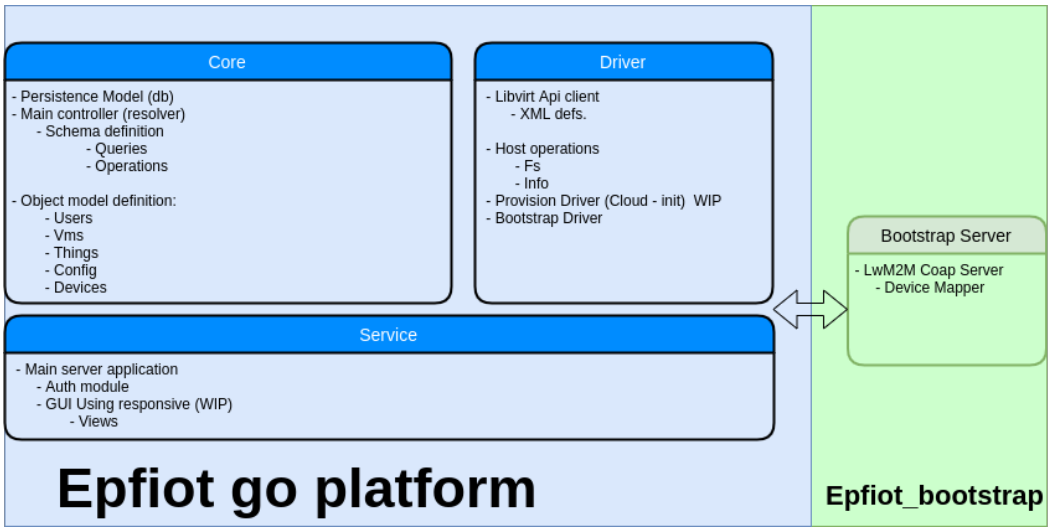

Figure 4.1: EPFIOT *overview.*

The following pages will show the implementation details taken for both parts.

4.1 The EPFIOT platform

The main EPFIOT platform is a program written in Go that makes up most of the EPFIOT Service. It provides the following features:

- A modern `http` API with basic user authentication.
- SQL as data model persistence.
- Infrastructure management with near baremetal performance with USB passthrough for accelerator computing.
- Basic infrastructure provisioning.
- IoT oriented.
- Management GUI.

Being a program written from scratch made it necessary to perform some architectural work in order to glue all the logic together. The result was the creation of some modules differentiated by the function they perform within EPFIOT, described next.

4.1.1 Driver Module

This module serves as an interface to different functionalities that EPFIOT has to deal with, mostly related to host communication or the appliance itself. Some of them are:

- Virtual machine management.
- Provisioning.
- File system and OS operations.
- UDP client.

Controller

The main element of this module is the Infrastructure Controller. This is the part of EPFIOT that communicates with the host and handles the virtual machines. To perform this secure communication, it uses SSH, which is compatible with `libvirt`, the virtualization API with which EPFIOT is able to control KVM and QEMU from the host side.

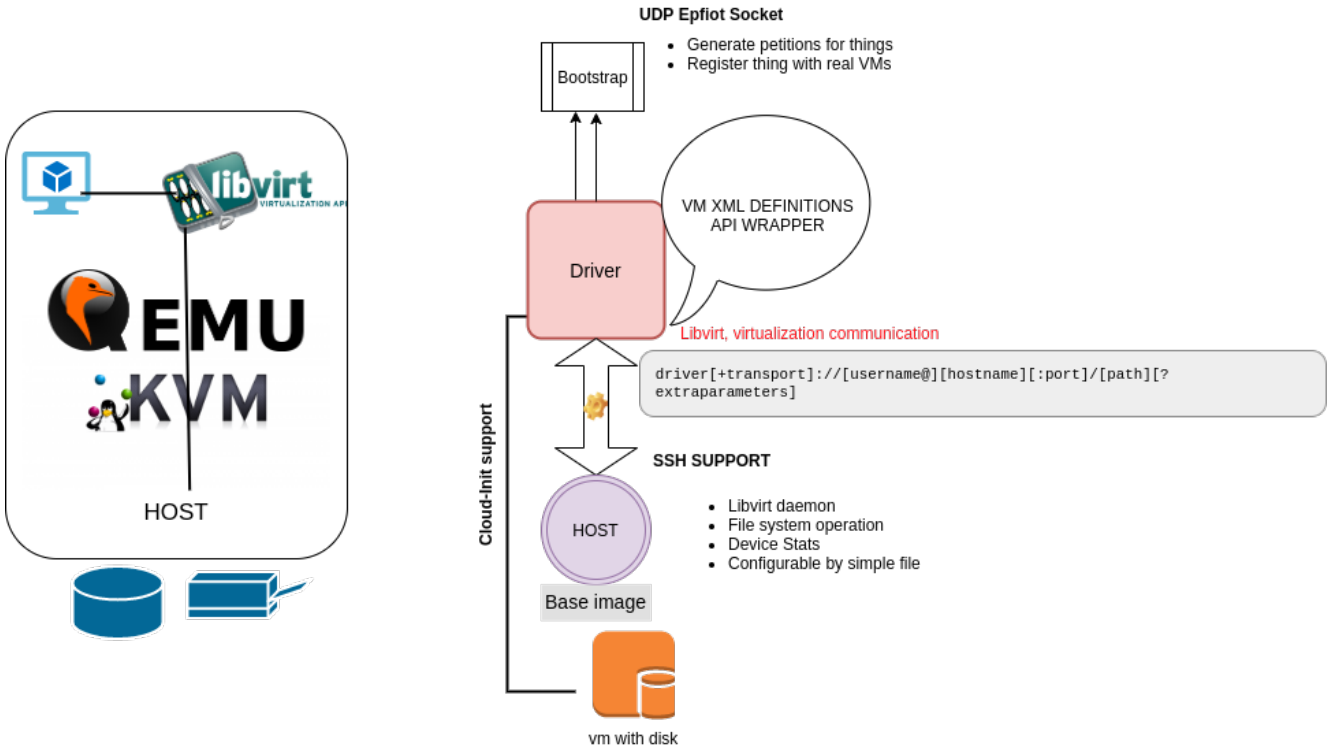

Figure 4.2: *Driver module overview.*

The Controller is written in a decoupled fashion and activated by a configuration file, which means that it is possible to write another controller to put it to work with Epfiot. This decision was made so that the project could evolve into newer forms of virtualization/containers in a near future without changing the rest of the functionality. To build the Controller, a Golang interface is used in order to allow Epfiot to work with certain infrastructure functionalities that the driver has to implement. The other Epfiot modules will work using the interface exposed by the controller. The interface that describes Epfiot that any driver has to follow is the next:

- Init: To perform any initialization task.
- Create: creates VM infrastructure.
- AttachDevice: Operation that allows to attach usb devices to the Vms.
- DettachDevice: Operation that allows to dettach usb devices to the Vm.
- List: List the provided infraestructure by the driver.
- Update: For change componentes of the Virtual machines.
- Shutdown: Turn off a Virtual machine.

- PowerOn: Turn on a Virtual machine.
- Destroy: Destroy a Virtual machine.
- ForceOff: Abruptly turns off the machine.
- ForceDestroy: Abruptly destroys the machine.

In the Epfiot project, the Libvirt driver is implemented, which exposing that interface will allow to orchestrate the infraestructure.

System operations (*Utils*)

Exclusively with the controller, it is not possible to perform all the operations related to the infrastructure. That is why the driver provides a number of functions in the form of utilities to support the controller. This part takes care of some IO operations such as creating/copying/deleting disks or getting host information.

Provisioning

Epfiot is able to perform some basic provisioning operations. This part of the driver module takes care or creating CD-ROMS with user settings (available via the Epfiot API). This CD-ROM is inserted into the machine at its deployment and thanks to Cloud-Init, it is able to perform actions such as creating users, installing packages, running scripts, etc.

As of today, in Epfiot, the provisioning part only provides the basic access credentials for the machine user and some basic networking; however, it is planned to expand the functionality in order to allow more operations.

UDP communication

Finally, the Driver module also handles Epfiot communications related to UDP. This is directly related to the bootstrapping part we will cover next.

The module acts as an UDP client to send certain information to the Epfiot Bootstrap process. This part of the module serves as an interface between Golang and C, something totally mandatory to use LWM2M implementations.

4.1.2 Service Module

This part of Epfiot is responsible for serving an interface to the user. Some of the purposes this module serves are as follows:

- Authentication and security.

- **HTTP API endpoint**.
- GUI.

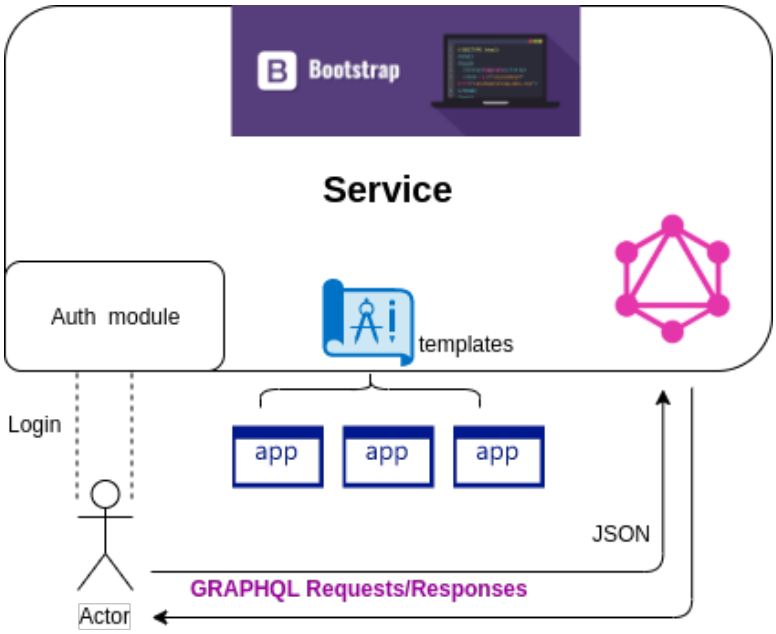

Figure 4.3: *Service module overview.*

Epfiot API

This module is responsible for exposing the operation API to the user; we use Golang `net/http` native library to provide a fast web service. As mentioned, this module, instead of using REST takes another perspective by using a Graphql framework. This allows Epfiot to have a single endpoint where it is able to expose all operations through one schema. This schema is described in Chapter 5.1.

With the user aware of this scheme, he/she is able to send JSON as a console to interact with Epfiot. This endpoint will respond with JSON objects depending on user specific requests.

Authentication

Epfiot is a multi-tenant application, which means that it is capable of registering users and interacting with them. For this reason it is necessary to authenticate with Epfiot. The service module handles this authentication by maintaining a secure session for the user.

In the current version of Epfiot, only a basic version of this module is implemented, which allows an identification of the user and maintaining their session.

GUI

To finish with this module we come to the part of the visual interface. The module has a visual system using the `html/css/javascript` templates offered by golang.

At the moment, the most functional part of this visual interface is the console. However, the logic that covers the rendering code is working and it is possible to make some graphic windows in a very simple way.

4.1.3 Core

Core is the most fundamental part of Epfiot. The Epfiot Core can be understood as the heart of the application. In this module, we find the code that controls the rest of the modules (like the one for the driver). In addition, the object model governing the entire application is defined here.

Some of the functions that this module performs are:

- Object definitions (model).
- Database layer for model persistence.
- Management of the data flow/operations of the entire application.

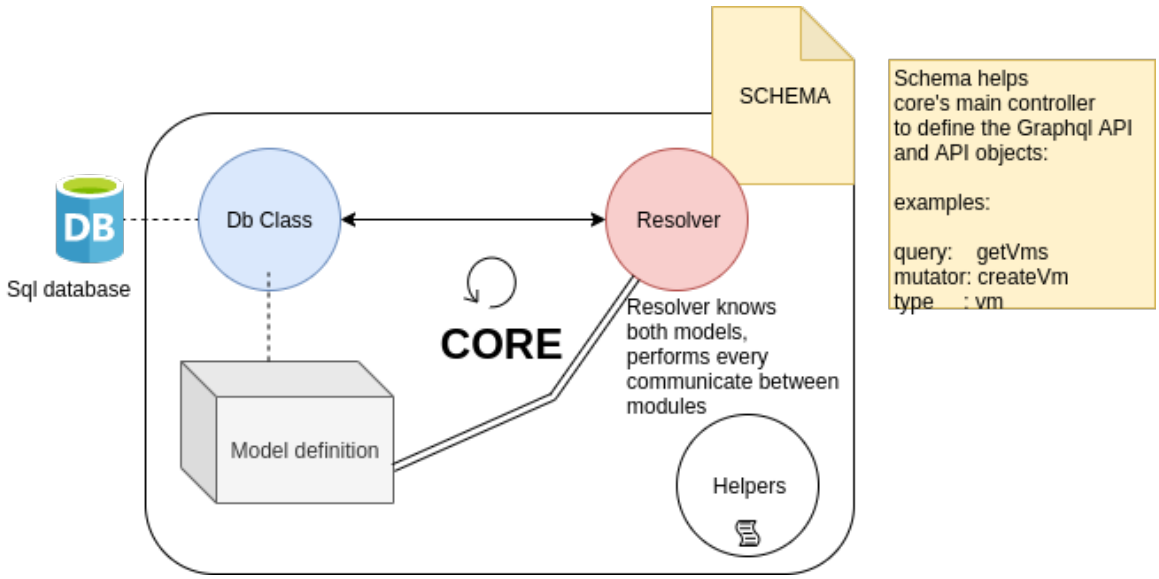

Figure 4.4: *Core overview.*

Resolver

Resolver is the class that handles the application. This code is in charge of receiving the actions dictated by the user through Graphql. It is capable of of calling other modules to perform specific actions. Inside this controller, we can find many data definitions that the user is able to use in first instance (see Chapter 5.1.1).

As a design decision, the Resolver saves a pointer to the instance of the Driver module with which EPFIOT is running. As mentioned, EPFIOT's architecture is decoupled and although at the moment there is only one driver for KVM, it is possible to implement any other as long as the interface is kept unchanged.

When the user makes an API call such as `createVm`, EPFIOT uses the service module to handle any `http` session, right after that Resolve dictates what action to take.

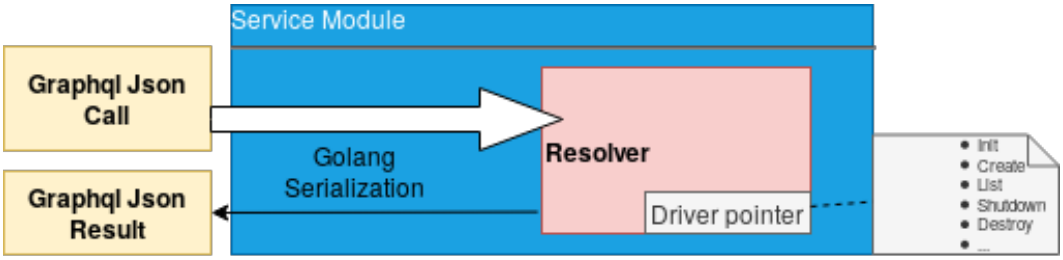

Figure 4.5: *Service - Resolver - Driver Flow*

Data persistence

As expected, one of the most important parts of the core is data persistence. EPFIOT uses a relational model to store data and infrastructure status. To take this process, Core has a class that is exclusively in charge of communicating with the database.

When changes are made to the model, Resolver is able to access this storage layer in order to allow data persistence.

Epfiot Model

We refer to the model as the series of defined objects and the relationships they form with each other that EPFIOT offers to the user in order to manage the infrastructure and IoT environment.

These objects that we can find inside EPFIOT are the following:

- **User**: The normal user entity that uses the application to know who owns the environment.

- **VM**: Representing the infrastructure, keeping information about the details of the machine and the status.
- **Devices**: Each of these objects represent the USB devices that are attached, storing information such as the bus/device number.
- **Things**: They represent the various IoT sensors deployed and their relationship to the infrastructure.
- **Config**: The only purpose of this object is to save instance initialization options that could be reused.

As can be observed, the Epfiot model is quite simple and useful for the user to build his/her environment without major drawbacks.

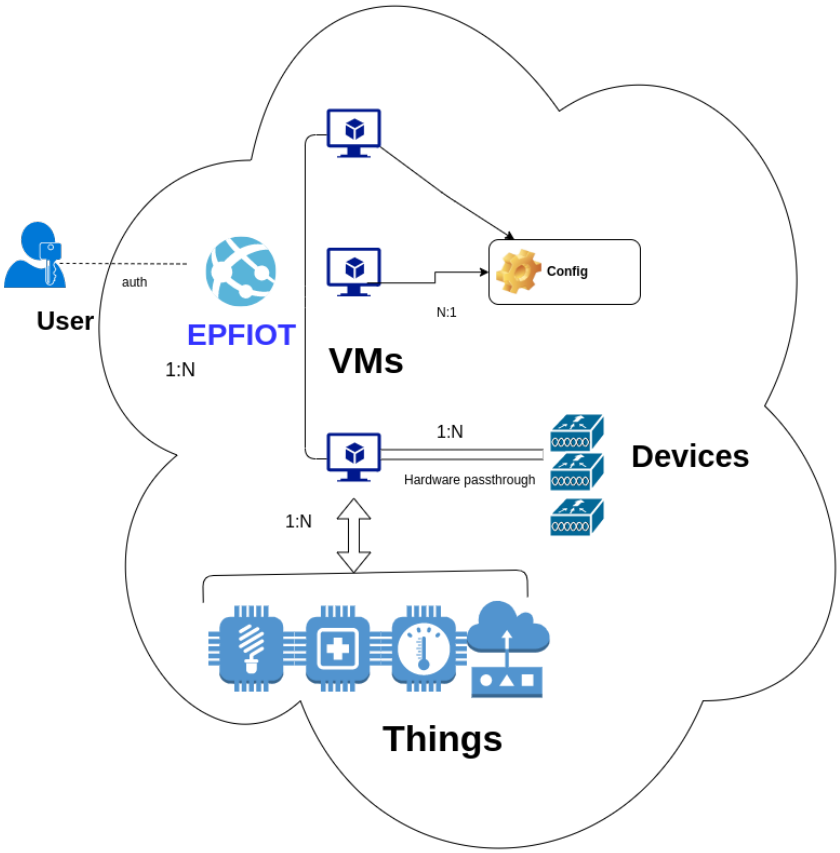

Figure 4.6: *Epfiot Model overview.*

4.2 EPFIOT Bootstrap

In the Core part, we have seen the underlying model of the application (section 4.1). Specifically, there is a part in this model called *"things"*. These objects not only serves to store information, but must actually be related to real sensors. To interact with those sensors, it was decided the use of LWM2M protocol, an application layer for IoT device management.

Integrating LWM2M into EPFIOT was not an easy task; is a relatively young protocol and there are still not many written implementations to use. Finally EPFIOT bootstrap was born a separate project made to meet the requirements of LWM2M and taking advantage of the Eclipse Foundation's Wakaama implementation.

This part of EPFIOT, as its name suggests, is based on the bootstrap service provided by LWM2M. A service with this implementation is launched in parallel to the main platform, and it is in charge of establishing relationships between the VMs and the *things*.

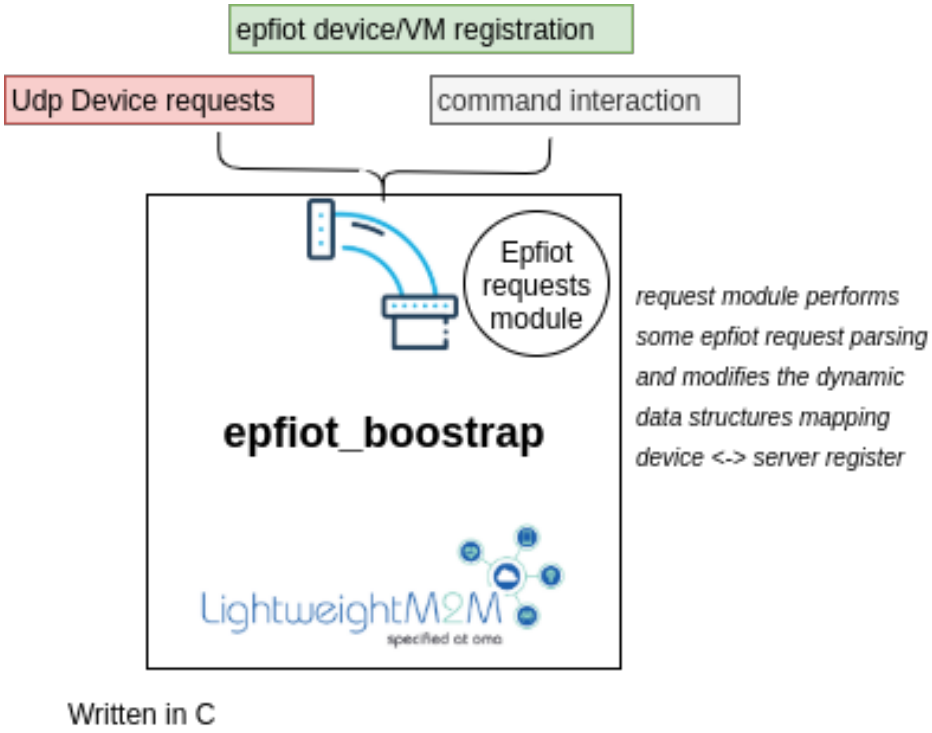

Figure 4.7: *Epfiot_bootstrap Overview*

This process listens to UDP packets and stores in memory the bootstrapping configuration for different sensors.

Some of the actions performed by EPFIOT bootstrap are:

- It has implemented a protocol on UDP built for Epfiot.
- It stores information about EPFIOT VMs and *things*.
- When a sensor asks for bootstrapping configuration, EPFIOT is able to configure it using its internal model.
- It notifies the main Platform that a client (sensor) has been registered and associated with a given VM.

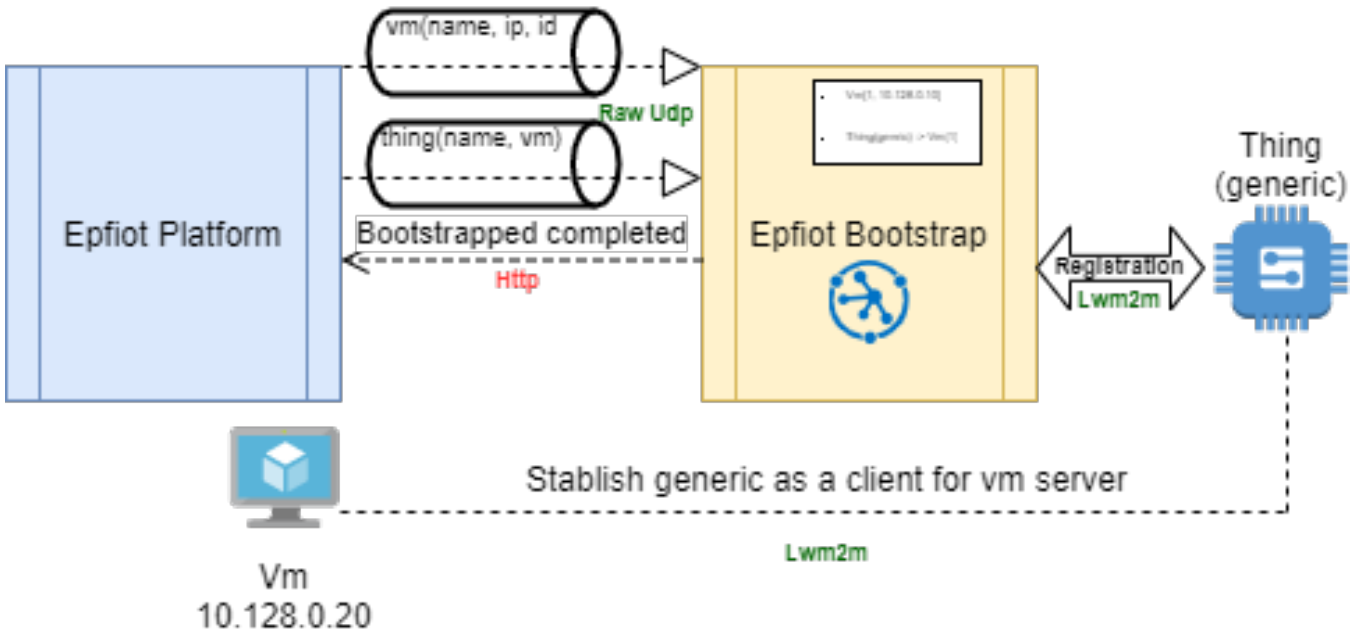

Figure 4.8: EPFIOT *bootstrap example.*

Chapter 5

The EPFIOT API

This chapter explains in detail the operations that EPFIOT has already implemented. As previously said, instead of using a classic REST API, EPFIOT takes a different approach, using graphql.

It is possible to see Graphql as a kind of new language; that is why it is is important to enumerate the main interface supported as well as the rules that must be followed in order to make requests in a properly manner. The service is created by defining types and fields on those types, and by providing functions for each field on each type.

We will proceed by defining the types and objects of the EPFIOT interface to further show the enabled operations of the project. It is useful to note that these operations or functions must be divided in two main groups, namely:

- *Queries*: these are the functions used by the client to request the data it needs from EPFIOT. It could be seen as Read-Only operations.
- *Mutations*: unlike queries, mutations are a kind of function that allows changes in the model. That means that it is possible to create, update or delete the data through them.

This text explains the possible ways to interact with Epfiot, however it is highly recommended to visit the Graphql documentation to know in depth details of this technology.

5.1 Types

To make the user interaction easier, EPFIOT is based on types; these types are used in two ways:

1. *As a parameter*: May be required when calling some operations.
2. *As a result*: It is very useful to know about how these types are constructed because when they are returned from an operation, the user is free to query any desired attribute.

These types are composed of several attributes. Attributes have their own type, primitively defined by graphql. Note that the following types are already native to graphql:

- *ID*: This type is used for unique identifiers.
- *String*: This is the UTF8 character sequence.
- *Int*: A signed 32-bit integer.
- *Boolean*: True or False.
- *List*: A list of values for a specific type.

Beyond the generic basic types exposed by Graphql, the custom types used in EPFIOT for performing operations are shown next. Next to each attribute, there is a column indicating the permission of the attribute:

- *Read*: if it can be read by the user as a result of some function.
- *Write*: if the user has the capability to modify it in any way.

5.1.1 User

This type is implemented to represent the user entity in the EPFIOT communication process. The main use of this type is for administrators.

Type User

Attribute	Type	Permissions	Description
id	ID	R	The user identifier
name	String	RW	The username
vms	[ID]	RW	The list of virtual machines owned by the user

Table 5.1: EPFIOT User type.

It is not common to perform any operation with this type; the handling of the users in EPFIOT is planned for future releases.

5.1.2 Hostdev

This type represents the physical device managed by EPFIOT. The most common way this type is used is to represent the USB accelerators attached to the host.

Type Hostdev

Attribute	Type	Permissions	Description
id	ID	R	The device identifier
bus	String	R	The bus number assigned by the Linux kernel
device	String	R	The device number assigned by the Linux Kernel
info	String	R	Text chain describing the USB context, normally the manufacturer. Allow us to identify the device.

Table 5.2: EPFIOT Hostdev type.

Keep in mind that Hostdev type is read-only. What does this mean? This type is essentially used to provide some useful information to the user in order to know which devices are available to the host. This means that no user should change this information, it depends exclusively on the thevices that are installed on the machine.

5.1.3 Thing

In an IoT context, *things* are devices that collect data from the environment. This type represents those devices.

Type Things

Attribute	Type	Permissions	Description
id	ID	R	The thing identificator
name	String	RW	friendly identificator, used for LWM2M
info	String	R	Text chain describing the thing

Table 5.3: Epfiot Thing type.

5.1.4 VM

This type represents a virtual machine, required for performing some basic operations such as create, update or get. This VM type will be used in order to pass the mandatory information to Epfiot.

Type Vm

Attribute	Type	Permissions	Description
id	ID	R	The virtual machine identificator, is autogenerated so this parameter is not required when creating a new virtual machine
owner	User	RW	Denotes the user to whom the virtual machine belongs
name	String	RW	The name of the machine (server as a friendly identifier)
base	String	RW	The name of the base image used for the vm creation. Epfiot will clone this image in order to create the machine
vcpu	Int	RW	The number of virtual cpus that the machine can run
memory	Int	RW	Amount of RAM in megabytes
state	string	R	Current state of the virtual machine
dev	[Hostdev]	RW	The list of devices (real hardware) attached to the machine
things	[Thing]	RW	The list of things related to the machine

Table 5.4: Epfiot User type.

The Vm type represents real machines that Epfiot has created, Attributes `dev` and `things` will show the relationships between VMs, USB accelerators and IoT devices.

5.1.5 Write-Only type parameters

We have seen the necessary types used by Epfiot API directly related to the model. However, to make the application easier to use for the user, some additional types have been created in order to serve as input values. These types will have write-only permissions in all their attributes and some of them will be mandatory.

Type ThingInput

Attribute	Type	Permissions	mandatory
name	String	W	yes
info	String	W	yes
Server	String	W	yes

Table 5.5: Epfiot ThingInput type.

Type ConfigInput

Attribute	Type	Permissions	mandatory
user	String	W	yes
password	String	W	yes

Table 5.6: Epfiot ConfigInput type table

This type has nothing to do with the ones mentioned above. This configuration will be applied when instantiating an Epfiot VM. At the moment, there are only two required fields: user and password, which will be automatically created in the underlying operating system in order to allow user control.

In the future, it is planned to improve this type to support more configurations, such as network or installation scripts.

Type VmInput

Attribute	Type	Permissions	mandatory
base	String	W	yes
name	String	W	yes
memory	Int	W	no
vcpu	Int	W	no
devIDs	[Int]	W	no
ThingIDs	[Int]	W	no
config	ConfigInput	W	no

Table 5.7: Epfiot VmInput type table

It is the same VM type seen earlier in write-only mode. It must be taken into account that to assign devices or things when instantiating a machine, these must exist previously and the user must know their identifier. However, it is possible to change these values using mutations.

5.2 Queries

Queries are a type of operation used to obtain information. Queries do not let the user to modify the internal model of Epfiot, meaning that resource creation or boot operation are beyond the scope of this section. Queries will be used when the user needs to query any kind of reading information such as active machines or devices associated with them. The reader should keep in mind that, thanks to Graphql, the user can view any attribute in an easy and simple way.

We describe the queries implemented in Epfiot below:

- **getUser(id: ID!): User**
  This operation gives an user object by passing it an id. Its use is intended for management purposes; however it is already implemented.
- **getUsers(): [User]**
  This operation returns all the users of the system, just like the previous one is intended for the administration of Epfiot.
- **getVm(id: ID!): Vm**
  Allows obtaining information about a specific machine already created. Epfiot will only permit to view those that are owned by the user.

- **getDev(id: ID!): Hostdev**
  Allows obtaining information about a specific host device. Host devices are created in program initialization.

- **getThing(id: ID!): Thing**
  Allows obtaining information about a specific *thing* already created. EPFIOT will only permit viewing those that are owned by the user.

- **getVms(): [Vm]**
  With this operation it is possible to view all VMs owned by the user.

- **getUsb(): [Hostdev]**
  In order to know which USB accelerators are plugged into the real host.

5.3 Mutations

These operations, as opposed to the queries, can change the model. This means that creation, startup and modification operations are allowed in this context, so the user should use them with care.

- **createVm(vm: VmInput!): Vm**
  Invoked for VM creation.

- **updateVm(vm: VmInput!): Vm**
  With this operation it is possible to change some aspects of a virtual machine, such as vcpu, memory or even adding devices.

- **deleteVm(userID: ID!, vmID: ID!): Boolean**
  Operation designed for EPFIOT administration. Allows to delete a VM of any user.

- **createThing(thing: ThingInput!): Thing**
  This operation allows to create *things*.

- **createThingVm(thing: ThingInput!, vmID: ID!): Thing**
  The previous operation has the invonvenience of creating a *thing* without being attached to anything. This operation creates the *thing* directly attached to one of the VMs.

- **attachThing(thingID: ID!, vmID: ID!): Boolean**
  Invoked to attach a thing already created to a VM.

- **attachDevice(devID: ID!, vmID: ID!): Boolean**
  Invoked to attach a device already created to a VM.

- **detachDevice(devID: ID!, vmID: ID!): Boolean**
  Detaches a specific device from a specific machine and releases it.

- **powerON(vmID: ID!): Vm**
  Turns on a VM ownwed by the user, changing to state running,

- **powerOFF(vmID: ID!): Vm**
  Power off a VM owned by the user, changing to state poweroff.

- **destroyVM(vmID: ID!): Boolean**
  Try to destroy a VM owned by the user. For this operation to be performed, the VM must already be switched off. Otherwise, the VM will simply shut down abruptly.

- **forceDestroyVM(vmID: ID!): Boolean**
  It forces the complete destruction of a VM even if it is in running state.

- **forceOFF(vmID: ID!): Boolean**
  It forces the shutdown of a machine.

Chapter 6

Examples and use cases

In this section, we expose a number of use cases that could be actually applied using the current status of the EPFIOT application. With these examples, a greater understanding is sought for the reader of how EPFIOT works. It is also intended to serve as a guide for beginners in the use of the application.

The proposed examples are as follows:

- **Scenario**: Before moving on to the examples, the proposed technical scenario is described.
- **Authentication and basic use**: This example explains the basics of EPFIOT. A user of the application logs in and creates a virtual machine.
- **Using VMs and accelerators**: A more advanced user decides to use EPFIOT and USB accelerators. He/she creates two virtual machines and decides to attach an accelerator to one of them with the purpose of comparing inference performance.
- **Taking care of things**: The user needs to set up a complete stack using EPFIOT: creating virtual machines, assigning accelerators, creating *things* and, in general, excersing all the capabilities of the system.

6.1 Scenario

We set up a simple yet descriptive scenario to deal with the examples in EPFIOT. Our main network will be a wireless WIFI network, managed by a standard access point/router with

DHCP enabled over the network 192.168.1.0/24. We install our NUC5i5RYK NUC in this network, assigning a static IP address in the network (192.168.1.141).

In the NUC, it is necessary to have a Linux installed and a KVM hypervisor. In Chapter 3.1 we detailed the necessary packages for an EPFIOT ecosystem from the host perspective. A series of installation scripts are available in EPFIOT main repository [28] to execute the following steps:

- Create the private network for EPFIOT (in our case, using `libvirt` routed feature), in this case 10.128.0.0/24.
- Create a virtual machine with the EPFIOT appliance with a static IP (10.128.0.141).
- Start the service.

The following figure sketches the scenario:

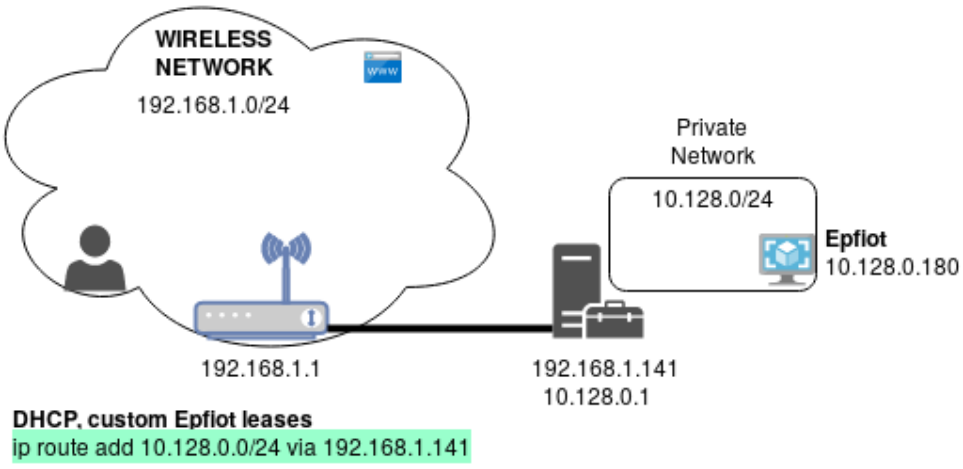

Figure 6.1: EPFIOT *experimental scenario.*

6.2 Authentication and basic use

EPFIOT is user-based: at the moment there is no public service dedicated to EPFIOT and the installation is private. For the alpha version (current) there are a number of example users already created to test the application.

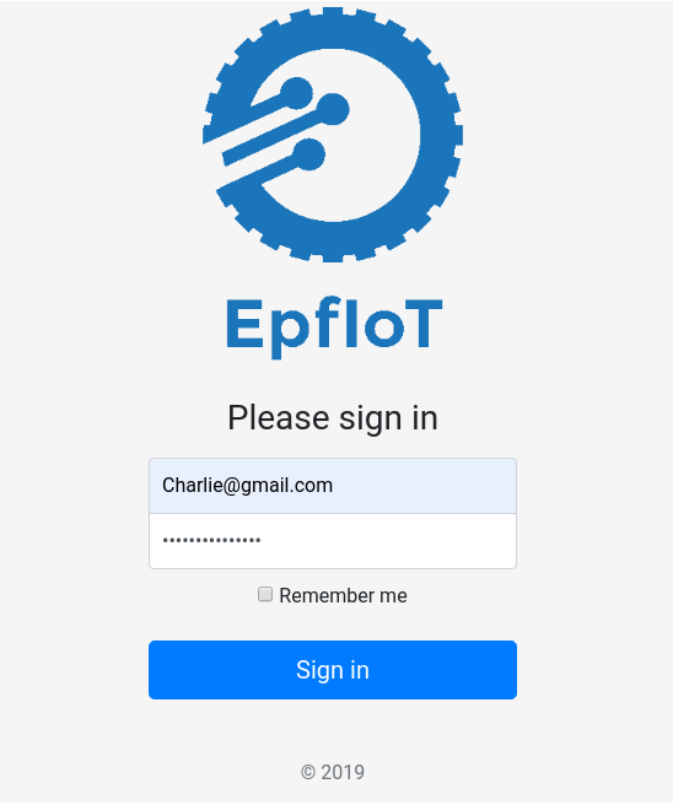

Figure 6.2: EPFIOT *login.*

Consider a hypothetical user `charlie@gmail.com`, that has access to the EPFIOT alpha and proceeds to enter into the application. EPFIOT provides an authentication system so entering a username and a password is mandatory to interact with the system.

Once logged in, the alpha version of EPFIOT provides a graphical Graphql console. The user `Charlie` can write in this console the operations already seen in Chapter 5.

This example is just a basic use of EPFIOT: the user will just create a machine in an edge environment. Therefore he/she will use the `createvm` operation adding a configuration to be able to enter the machine, see Figure 6.3.

As can be observed, the user is asking for specific return values, such as the status of the machine, its name or even the owner's ID. Upon machine creation, the user is ready to perform basic operations like, for example, turnin the VM on:

```
mutation  powerOn(vmID:1)
```

Although the command itself offers a clear feedback about whether the operation has been successfully completed or not, the user probably needs to know more information about

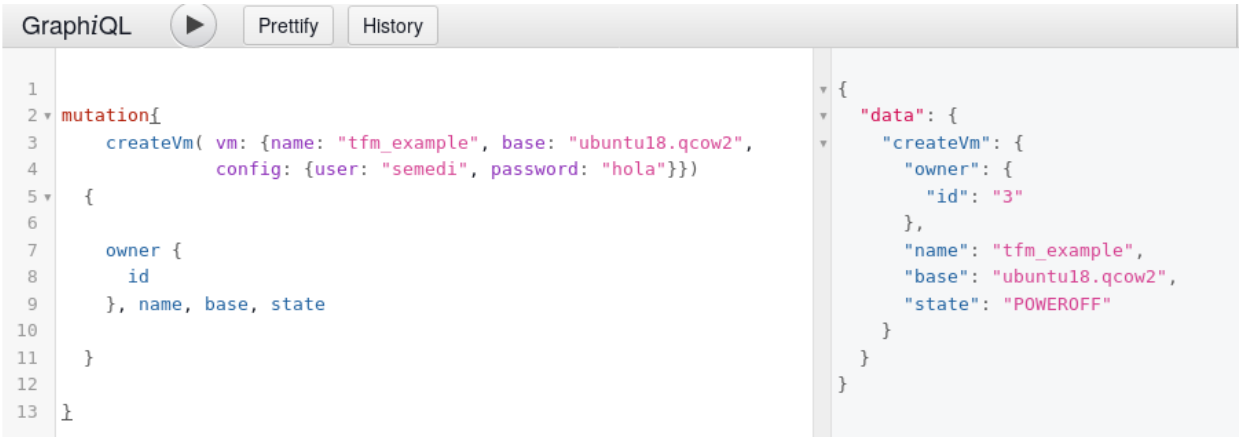

Figure 6.3: *Vm creation.*

his/her machine. Charlie uses the `getvms` command to see which is the current status, see Figure 6.4.

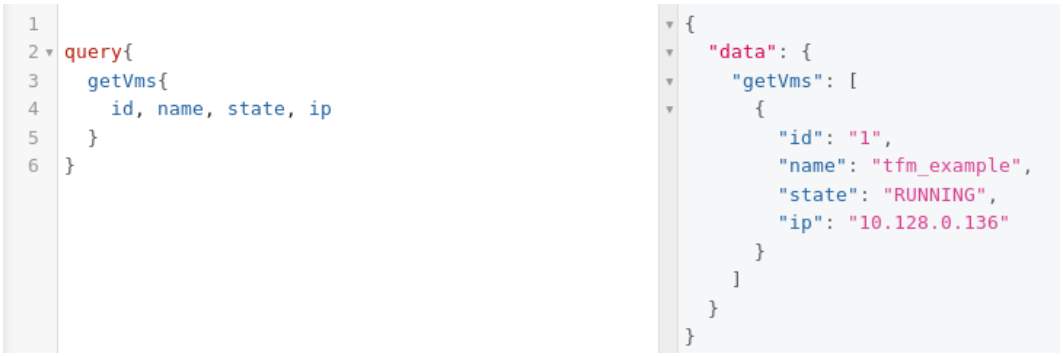

Figure 6.4: *VM creation.*

At this point, Charlie is able to use his/her new machine in the edge environment. It is important to note that the machine is accessible through SSH, using as credentials those which have already been provided in the creation of the machine. In this example, the user will build everything on his own and only uses EPFIOT to create a basic infrastructure.

6.3 Using VMs and accelerators

In this example, Charlie will use one of the most important features of EPFIOT: using real hardware accelerators attached to the machine. To do this, Charlie will use two machines:

- The first machine is the same as that used in the first example. For this machine

(10.128.0.136) Charlie will attach the Google Coral USB accelerator to improve inference performance through its use.

- The user will create a second machine (just like the first example), this time without any attached device. The purpose is to see if there was really a deterioration in speed compared to the previous test.

First of all, Charlie should check which USB devices are available on the host; for this, EPFIOT provides the `getusb` operation, which returns a list of the physical devices, see Figure 6.5.

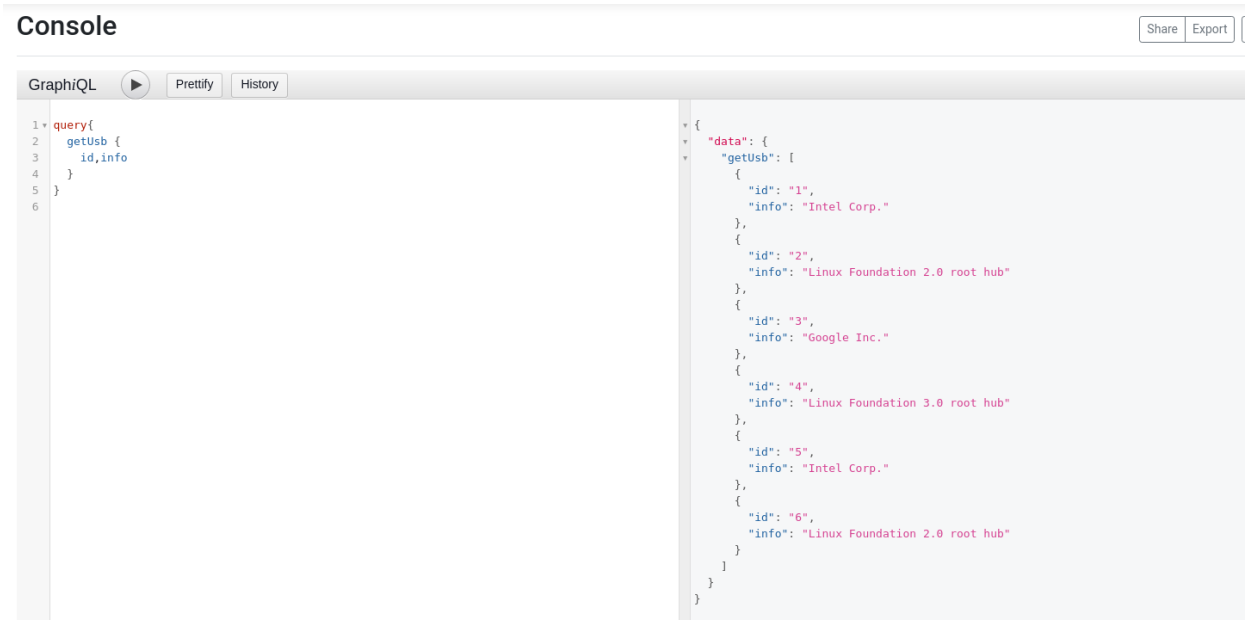

Figure 6.5: `getusb` *operation.*

In this case, we know that Google is the manufacturer of the device; therefore, our USB accelerator has ID 3 as seen in the Figure 6.5. Once the device ID is known, we need to find the ID associated to our machine as well. To do this we can use the `getVms` operation, as done in Figure 6.4, where we showed some interesting information of the machine such as its ID, status or the IP itself.

If we look at the **attachDevice** header in Chapter 5.3, we observe that it needs two arguments, which are the IDs already obtained, see Figure 6.6.

EPFIOT quickly reports that the operation has been successfully completed by returning a boolean value; however, Charlie desires to check it himself. Figure 6.7 shows the command that Charlie executed on his command console in order to check that the machine is actually running; it has the user and password indicated by EPFIOT and it has a USB accelerator attached.

Figure 6.6: `attachdevice` *operation.*

```
❯ sshpass -p "hola" ssh semedi@10.128.0.136 lsusb
Bus 002 Device 002: ID 18d1:9302 Google Inc.
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Figure 6.7: *Charlie checking the machine*

As seen, the machine is fully operational and has the Google coral installed.

Now that the first machine is ready, Charlie proceeds to create another machine just like the first one, this time without attaching any additional device. The result is a machine using the Google coral accelerator and another one that will use exclusively a virtual CPU.

Using Google Coral Accelerator

Charlie decides to test if he/she can really use the Google Coral accelerator from a virtual machine. To do this, he/she runs a sample code for Google Coral on the machine with the attached device and gets the results shown in Figure 6.8.

```
root@ubuntu:/coral/tflite/python/examples/classification# python3 classify_image.py --model models/mob
ilenet_v2_1.0_224_inat_bird_quant_edgetpu.tflite --labels models/inat_bird_labels.txt --input images/p
arrot.jpg
----INFERENCE TIME----
Note: The first inference on Edge TPU is slow because it includes loading the model into Edge TPU memo
ry.
15.3ms
4.6ms
4.7ms
4.6ms
4.6ms
-------RESULTS--------
```

Figure 6.8: *Testing VMs using Google Coral.*

Charlie obtains pretty good results: around 4.6 ms per inference.

Using Virtual CPU

Now, Charlie is preparing to do the same test using the second machine without the aid of any additional device, see Figure 6.9.

```
root@ubuntu:/coral/tflite/python/examples/classification# python3 classify_image.py   --model models/m
obilenet_v2_1.0_224_inat_bird_quant.tflite   --labels models/inat_bird_labels.txt   --input images/par
rot.jpg
----INFERENCE TIME----
Note: The first inference on Edge TPU is slow because it includes loading the model into Edge TPU memo
ry.
249.7ms
246.8ms
247.1ms
247.3ms
247.0ms
-------RESULTS--------
Ara macao (Scarlet Macaw): 0.78516
```

Figure 6.9: *Test vanilla VM.*

It is easy to observe the difference in the results (perfromance); Charlie gets a speed of 247ms, dramatically reducing the obtained results in the previous test.

With this test, Charlie has learned that EPFIOT's feature of attaching real devices is useful in certain occasions, such as those that could happen in an edge environment.

6.4 Taking care of *things*

Once Charlie knows about the real device management feature, he/she decides to deploy some sensors with which he/she will build an Internet of Things stack using the infrastructure already created. After the previous test, the user created two machines; quickly he/she realizes that one of them is no longer needed, so it is deleted as seen in Figure **??**

```
mutation{
  forceDestroyVM(vmID:1)
}
```

```
{
  "data": {
    "forceDestroyVM": true
  }
}
```

Figure 6.10: *Destroying a machine.*

Forcing a destroy operation 5.3 allows a user to abruptly remove a machine.

Continuing only with one machine, the user realizes that he/she does not remember the name given to the machine, so Charlie needs to determine it, together with its IP, memory and Virtual CPUs. Thanks to Graphql, EPFIOT allows using queries asking only for the desired values, see Figure 6.11.

Looking at EPFIOT's log, the user observes that something happened to the machine. To handle the sensors, EPFIOT registers the infrastructure on its bootstrapping server. Therefore, the "example2" machine is ready to be used with new sensors, see Figure 6.12.

```
query{
  getVms{
     name, ip, memory, vcpu
  }
}
```

```
{
  "data": {
    "getVms": [
      {
        "name": "example2",
        "ip": "10.128.0.169",
        "memory": 256,
        "vcpu": 1
      }
```

Figure 6.11: *Specific* `getvm` *operation.*

```
> petition from epfiot:
81 bytes received from [::ffff:127.0.0.1]:56386
73 65 72 76  65 72 7C 69  64 3D 31 2C  75 72 69 3D   server|id=1,uri=
63 6F 61 70  3A 2F 2F 31  30 2E 31 32  38 2E 30 2E   coap://10.128.0.
31 36 39 3A  35 36 38 33  2C 62 6F 6F  74 72 73 74   169:5683,bootrst
61 70 3D 6E  6F 2C 6C 69  66 65 74 69  6D 65 3D 33   ap=no,lifetime=3
30 30 2C 73  65 63 75 72  69 74 79 3D  4E 6F 53 65   00,security=NoSe
63                                                   c
```

Figure 6.12: Bootstrap *server log.*

This EPFIOT functionality allows the user to register sensors directly within the infrastructure. EPFIOT saves the device information in its bootstrap server, so that as soon as the device enters into the network it can be configured with the setting of the newly created server automatically. To do this, Charlie creates a new test device (a *thing*) on EPFIOT, see Figure 6.13.

```
mutation{
  createThing(thing:{
    name: "prueba",
    info:"prueba",
    server: "example2",
  }) {
    id
  }
```

```
{
  "data": {
    "createThing": {
      "id": "1"
    }
  }
}
```

Figure 6.13: *Creating a* thing.

The *prueba* sensor is now linked to the `example2` machine, at least from the point of

view of the EPFIOT mode (see Figure 6.14). For the device to be automatically configured, it must be first deployed in the network where EPFIOT is located.

Following the Figure 6.1, Charlie deploys his/her new sensor in the wireless network (192.168.1.0/24). Remember that, since the router has a table for the 10.128.0.0/24 network, it is possible to access the EPFIOT private network and reaching the allocated VMs.

```
petition from epfiot:
49 bytes received from [::ffff:127.0.0.1]:41396
65 6E 64 70  6F 69 6E 74  7C 4E 61 6D  65 3D 70 72   endpoint|Name=pr
75 65 62 61  2C 44 65 6C  65 74 65 3D  2F 30 2C 44   ueba,Delete=/0,D
65 6C 65 74  65 3D 2F 31  2C 53 65 72  76 65 72 3D   elete=/1,Server=
31                                                   1
```

Figure 6.14: EPFIOT *bootstrap device registration.*

With EPFIOT ready, Charlie turns on his/her new LWM2M sensor into the network and watches the generated log as that in Figure 6.15.

```
Bootstrap request from "prueba"
Sending DELETE /0 to "prueba" OK.
8 bytes received from [::ffff:127.0.0.1]:36438
64 42 92 92  92 92 2C 88   dB....,.

 Received status 2.02 (COAP_202_DELETED) for URI /0 from endpoint prueba.
Sending DELETE /1 to "prueba" OK.
8 bytes received from [::ffff:127.0.0.1]:36438
64 42 92 93  93 92 2E 88   dB......

 Received status 2.02 (COAP_202_DELETED) for URI /1 from endpoint prueba.
Sending WRITE /0/1 to "prueba" OK.
8 bytes received from [::ffff:127.0.0.1]:36438
64 44 92 94  94 92 2E 88   dD......

 Received status 2.04 (COAP_204_CHANGED) for URI /0/1 from endpoint prueba.
Sending WRITE /1/1 to "prueba" OK.
8 bytes received from [::ffff:127.0.0.1]:36438
64 44 92 95  95 92 2E 88   dD......

 Received status 2.04 (COAP_204_CHANGED) for URI /1/1 from endpoint prueba.
Sending BOOTSTRAP FINISH  to "prueba" OK.
15 bytes received from [::ffff:127.0.0.1]:36438
64 44 92 96  96 92 2E 88  FF 66 69 6E  69 73 68  dD.......finish

 Received status 2.04 (COAP_204_CHANGED) for URI / from endpoint prueba.
```

Figure 6.15: EPFIOT *psychical sensor detected.*

EPFIOT discovers this new sensor *prueba* and proceeds to send its new bootstrapping configuration. In the alpha phase EPFIOT follows the following steps:

- Delete /0 object.
- Delete /1 object.
- Write the *master* server configuration (in this case 10.128.0.169).

To finish his test, Charlie checks if anything has happened on the deployed server, getting the results in Figure 6.16.

```
❯ sshpass -p "hola" ssh semedi@10.128.0.169 lwm2mserver
> 127 bytes received from [::ffff:10.128.0.1]:38817
44 02 A2 1F  2F DB 85 16  B2 72 64 11  28 39 65 70   D.../....rd.(9ep
3D 70 72 75  65 62 61 06  6C 74 3D 33  30 30 09 6C   =prueba.lt=300.l
77 6D 32 6D  3D 31 2E 30  0B 6D 61 63  3D 75 6E 6B   wm2m=1.0.mac=unk
6E 6F 77 6E  FF 3C 2F 3E  3B 72 74 3D  22 6F 6D 61   nown.</>;rt="oma
2E 6C 77 6D  32 6D 22 3B  63 74 3D 31  31 35 34 33   .lwm2m";ct=11543
3B 68 62 2C  3C 2F 31 2F  31 3E 2C 3C  2F 33 2F 30   ;hb,</1/1>,</3/0
3E 2C 3C 2F  33 33 30 33  2F 30 3E 2C  3C 2F 33 33   >,</3303/0>,</33
30 33 2F 31  3E 2C 3C 2F  33 33 31 32  2F 30 3E      03/1>,</3312/0>
```

Figure 6.16: *Deployed VM receiving information about a new device.*

The *prueba* device is now attached to the virtual machine previously created. From this point on, and thanks to the EPFIOT infrastructure, Charlie can continue interacting remotely with the new sensor.

Since EPFIOT has configured the *prueba* sensor, Charlie can also see which devices are active (they are in the network and have been bootstrapped by the application), see Figure 6.17.

```
1
2 query{
3   getVm(id: 1) {
4     name, things {
5       id, name
6     }
7   }
8 }
```

```
{
  "data": {
    "getVm": {
      "name": "example2",
      "things": [
        {
          "id": "1",
          "name": "prueba"
        }
      ]
    }
  }
}
```

Figure 6.17: *The bootstrapped* thing.

Chapter 7

Conclusions and future work

7.1 Conclusions

This work aimed at studying the feasibility of deploying a dynamic infrastructure following a cloud-like model, adapted to edge computing, with the associated restrictions in terms of compute resources. Therefore, we decided to implement and design a lightweight, energy- and compute-saving appliance. The result is EPFIOT, a complete edge-computing infrastructure that exhibits the following functionalities:

- Web multi-tenant application, following the basic cloud principles with a data storage model and a visual interface.
- Infrastructure management with near-baremetal performance and provisioning system.
- Ability to use real hardware accelerators, attaching them dynamically to the infrastructure.
- Oriented to the IoT paradigm, using new protocols such as LWM2M to provide an easy handling of the sensors within the environment.

With these features in mind, it can be said that the objectives have been satisfactorily achieved. The use cases illustrated in Chapter 6 demonstrate that it is possible to use the platform in a real edge environment, satisfying the basic needs of a hypothetical user.

Finally, it should be noted that a hypervisor like KVM is undoubtedly effective for this kind of project, but emulators such as QEMU should be replaced in order to achieve a lower overhead and lighter guests. However, it is safe to say that by using this type of technology a quite useful virtual environment can be guaranteed for the end user.

7.2 Future Work

Developing a product like EPFIOT has been a hard work due to the complexity involved. To be able to build this type of platform, the developer needs to deal and take many factors into account, as the application is large and many extra concepts, such as security, have to be properly considered.

With this in mind, EPFIOT achieved the basic objectives for which the project was designed; however some tasks can be considered in order to improve the project, and managed as future work:

- **Stability & maintenance**: Like any application, EPFIOT also requires maintenance and bug hunting.
- **Improve security**: For the development of the project, some tradeoffs in terms of security have been made and additional security concerns (such as the use of encryption) should be in place in a stable version of the product.
- **Better infra provisioning**: EPFIOT allows a basic provisioning mechanism for the machines; it would be useful to improve this mechanism.
- **Configurable thing bootstrap**: EPFIOT performs a series of static steps to configure devices; thanks to LWM2M, this part can be easily customized by the user in the future.
- **Expand the graphic interface**: At the moment, only a basic part of the GUI is developed, as is was not within the scope of the project to provide a full administration GUI for EPFIOT.

It is important to mention that EPFIOT leaves open the possibility of developing more than one virtualization driver for the application. This means that it is possible to expand the project with some new technology in a straightforward manner. Therefore, a good option in the future could be to implement another driver and compare performance with the current one.

Bibliography

[1] Vangie Beal. Cloud service Term. https://www.webopedia.com/TERM/C/cloud_services.html. Accessed: 2020-19-03.

[2] Roberto Morabito. Virtualization on internet of things edge devices with container technologies: A performance evaluation. *IEEE Access*, 5:8835–8850, 2017.

[3] Virtualization term. https://www.techadvisory.org/2014/05/why-choose-virtualization/. Accessed: 2020-20-03.

[4] Diane Barrett and Gregory Kipper. *Virtualization and Forensics*, volume 1st Ed. Syngress, 2010.

[5] M. Aazam and E.Huh. Fog computing and smart gateway based communication for cloud of things. *Proceedings - 2014 International Conference on Future Internet of Things and Cloud, FiCloud 2014*, pages 464–470, 2014.

[6] C K. Dolui and Kiraly. Towards multi-container deployment on iot gateways. *arXiv Computer Science*, 2018.

[7] What's a Linux container? https://www.redhat.com/en/topics/containers/whats-a-linux-container. Accessed: 2020-23-03.

[8] C.Hong and B.Varghese. Resource management in fog/edge computing: A survey. *Computing Research Repository*, 2018.

[9] J.Wang W.Zhang T.Li C.Li, Y.Xue. Edge-oriented computing paradigms. *Computing Research Repository*, 51:1–34, 2018.

[10] Uses for VMs vs Uses for Containers. https://www.backblaze.com/blog/vm-vs-containers/. Accessed: 2020-24-03.

[11] Openstack official page. https://www.openstack.org/software. Accessed: 2020-24-03.

[12] Wikipedia Cloudlet. https://en.wikipedia.org/wiki/Cloudlet. Accessed: 2020-24-03.

[13] Kiryong Ha and Mahadev Satyanarayanan. Openstack++ for cloudlet deployment. pages 1–24, 2015.

[14] Alfonso A. Carrillo Ignacio M. LLorente Rubén S. Montero, Elisa Rojas. Extending the cloud to the network edge. *IEEE Computer*, 50:91–95, 2017.

[15] OneEdge main page. https://oneedge.io/. Accessed: 2020-25-03.

[16] AWS Outpost. https://aws.amazon.com/es/outposts/. Accessed: 2020-24-03.

[17] Intel Vt-x Concept. https://www.hardwaresecrets.com/everything-you-need-to-know-about-the-intel-virtualization-technology/. Accessed: 2020-25-03.

[18] KVM kernel module. https://www.linux-kvm.org/page/Main_Page. Accessed: 2020-25-03.

[19] Live migrating QEMU-KVM. https://developers.redhat.com/blog/2015/03/24/live-migrating-qemu-kvm-virtual-machines/. Accessed: 2020-25-03.

[20] Libvirt official page. https://libvirt.org/index.html. Accessed: 2020-25-03.

[21] Libvirt virtual networking comparison. https://wiki.libvirt.org/page/VirtualNetworking. Accessed: 2020-25-03.

[22] Nuc Board Specifications. https://ark.intel.com/content/www/us/en/ark/products/83254/intel-nuc-kit-nuc5i5ryk.html. Accessed: 2020-25-03.

[23] Tech Specs for google coral and overview. https://coral.ai/products/accelerator. Accessed: 2020-25-03.

[24] Libvirt RPC infraestructure. https://libvirt.org/internals/rpc.html#internals. Accessed: 2020-25-03.

[25] Olaf Hartig and Horge Perez. An initial analysis of facebook's graphql languaje. 2017.

[26] Y.Zhang S.Akhouri S.Slovetskiy, P.Magadevan. Lightweight m2m 1.1: Managing non-ip devices in cellular iot networks. 2018.

[27] C.Deshpande V.Lakkundi S.Rao, D.Chendanda. Implementing lwm2m in constrained iot devices. *IEEE Access*, 2015.

[28] Epfiot Installation scripts. https://github.com/Semedi/epfiot/tree/master/scripts. Accessed: 2020-23-05.