UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
PROGRAMA DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA

Leonardo José Held

AN ARCHITECTURE FOR CONTEXT-AWARE DEPLOYMENT AND MONITORING OF IOT APPLICATIONS USING KUBERNETES

Florianópolis, Santa Catarina – Brasil
2023

Leonardo José Held

AN ARCHITECTURE FOR CONTEXT-AWARE DEPLOYMENT AND MONITORING OF IOT APPLICATIONS USING KUBERNETES

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Engenharia Eletrônica da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Engenharia Eletrônica.
Orientador(a): Praveen Mohanram,

Florianópolis, Santa Catarina – Brasil
2023

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina.
Arquivo compilado às 11:20h do dia 23 de fevereiro de 2023.

Leonardo José Held
An Architecture for Context-Aware Deployment and Monitoring of IoT Applications Using Kubernetes / Leonardo José Held; Orientador(a), Praveen Mohanram, - Florianópolis, Santa Catarina - Brasil, 01 de março de 2023.
70 p.

Trabalho de Conclusão de Curso - Universidade Federal de Santa Catarina, EEL - Departamento de Engenharia Elétrica e Eletrônica, CTC - Centro Tecnológico, Programa de Graduação em Engenharia Eletrônica.

Inclui referências

1. Virtualização, 2. Internet das Coisas, 3. Sistemas Embarcados, I. Praveen Mohanram, II. Programa de Graduação em Engenharia Eletrônica III. An Architecture for Context-Aware Deployment and Monitoring of IoT Applications Using Kubernetes

CDU 02:141:005.7

Leonardo José Held

AN ARCHITECTURE FOR CONTEXT-AWARE DEPLOYMENT AND MONITORING OF IOT APPLICATIONS USING KUBERNETES

Este(a) Trabalho de Conclusão de Curso foi julgado adequado(a) para obtenção do Título de Bacharel em Engenharia Eletrônica, e foi aprovado em sua forma final pelo Programa de Graduação em Engenharia Eletrônica do EEL – Departamento de Engenharia Elétrica e Eletrônica, CTC – Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brasil, 01 de março de 2023.

Prof. Fernando Rangel de Sousa, Dr.
Coordenador(a) do Programa de Graduação
em Engenharia Eletrônica

Banca Examinadora:

M.Praveen

Praveen Mohanram, Me.
Orientador(a)
Fraunhofer-Institut für
Produktionstechnologie – IPT

Prof. Richard Demo Souza, Dr.
Universidade Federal de Santa Catarina

Prof. Raimes Moraes, Dr.
Universidade Federal de Santa Catarina

Dedicated to my cat, Sid.

"Where the fear has gone there will be nothing. Only I will remain."

Frank Herbert

RESUMO

Uma característica marcante dos projetos de sistemas embarcados nos últimos anos é a adoção de tecnologias de nuvem como containerização e conexão com serviços computacionais de escala a fim de gerenciar frotas de dispositivos de Internet das Coisas. Este trabalho propõe a continuação da tendência de uso de tecnologias originárias na nuvem em sistemas embarcados, com o estudo e implementação de uma arquitetura de gerenciamento de frota utilizando Kubernetes. O trabalho proporciona uma recapitulação histórica do uso e desenvolvimento de virtualização leve em sistemas rodando Linux, além de propor e implementar uma arquitetura de software utilizando orquestrador, apoiada por casos de uso com o objetivo de ser um primeiro passo para projetos distribuídos de Internet das Coisas.

Palavras-chaves: Virtualização. Internet das Coisas. Sistemas Embarcados.

RESUMO EXPANDIDO

INTRODUÇÃO

O mercado de dispostivos embarcados no seguimento de Internet das Coisas terá mais de 29 bilhões de dispositivos conectados até 2023 (ATZORI; IERA; MORABITO, 2010). Diversas aplicações desses sistemas embarcados possuem requisitos de computação distribuída, ou de serem interconectados com sistemas de maior processamento e baixa latência, os chamados *sistemas de computação de borda*.

Apesar da necessidade desse tipo de sistema, problemas naturais adventos da arquitetura complexa de sistemas distribuídos podem surgir durante o projeto, como por exemplo a complexidade em monitoramento, atualização e reversão remotas. Visando suprimir esses problemas, uma característica marcante dos últimos anos em projetos de sistemas embarcados, principalmente aqueles baseados em Linux ou *BSDs, é o avanço do uso de tecnologias que se originaram em ambientes de nuvem (CHIMA OGBUACHI et al., 2020). Exemplos incluem tecnologias de virtualização leve e o próprio uso de serviços de nuvem que servem de controle ou integração para frotas de dispositivos de Internet das Coisas.

OBJETIVOS

A comodização de serviços de nuvem com ofertas de computação massiva e sob demanda é impulsionada por tecnologias como orquestração de containers, que oferecem escalabilidade e alta disponibilidade (ou seja, sem tempo de inatividade) na implementação de serviços de software, possibilitado arquitetura mais distribuídas, como a de microsserviços. Nesse sentido, existe um argumento forte para o uso de não só tecnologias de virtualização leve como também o uso de orquestradores em sistemas de Internet das Coisas.

O maior desafio na aplicação de orquestradores em sistemas de Internet das Coisas é a limitação do número de variáveis que o sistema orquestrador leva em consideração, já que a tecnologia é nativa de ambientes de nuvem, onde poucas métricas importam no escalonamento de cargas de software (CHIMA OGBUACHI et al., 2020).

Visando continuar a tendência de utilizar tecnologias de nuvem para gerir sistemas embarcados distribuídos, o presente trabalho pretende fornecer um ponto de partida para projetos de sistemas embarcados de Internet das Coisas que desejem utilizar um orquestrador como sistema de gerencia de frota, criando as aplicações necessárias para amendar a limitação supracitada dos orquestradores atuais.

Como objetivos específicos também temos:

- Fornecer uma visão geral sobre os desenvolvimentos históricos sobre tecnologias de virtualização e orquestração.

- Desenvolver uma arquitetura de referência para projetos de sistemas embarcados que estejam avaliando a utilização de tecnologias de virtualização e orquestração.
- Descrever o desenvolvimento de uma implementação da arquitetura de referência.
- Desenvolver um sistema de monitoramento a ser agregado ao orquestrador, a fim de suprir a falta de parâmetros particulares de sistemas embarcados nos orquestradores atuais.
- Realizar testes de performance e apresentar usos de caso possíveis com a arquitetura desenvolvida.

METODOLOGIA

Na construção de uma arquitetura de software, inicialmente foram identificados os componentes básicos de um sistema de Internet das Coisas distribuído, e postos em ordem de capacidade computacional em formato de camadas. São esses: Camada de Nuvem Pública (computação por demanda de servidores como Amazon Web Services e Google Cloud Platform) e Camada de Nuvem de Borda (computadores locais de alta performance), que compõem a parte de Tecnologia de Informação do sistema e Camada de Borda (dispositivos embarcados rodando Linux, como *Raspberry Pis*) e Camada de Sensores Inteligentes (microcontroladores equipados com sensores e atuadores), que compõem a parte de Tecnologia Operacional do sistema. Cada camada é composta por um número variável de dispositivos denominados de *nó*.

Também foi posto que o orquestrador deve ser o componente central e oferecer uma camada coesa de abstração sobre todas as outras camadas. Dessa forma, dois tipos de comunicação foram estabelecidos: Dados de Monitoramento e Comandos de Implantação ou Reconfiguração dos dispositivos que compõe cada camada.

Como anteriormente citado, foram identificadas falhas na capacidade do orquestrador de tomar em consideração e obter dados customizados necessários para o monitoramento de frotas de Internet das Coisas. Visando suprir essa deficiência, os Dados de Monitoramento são de responsabilidade de um serviço de monitoramento de frota a ser desenvolvido nesse trabalho. Os Comandos de Implantação ou Reconfiguração são responsabilidade do orquestrador.

O serviço de monitoramento é composto em duas partes: Agente de Monitoramento e Servidor de Monitoramento. Os Agentes são executados em cada nó, e enviam diversas informações sobre o contexto do dispositivo até o servidor, que insere as informações numa base da dados que funciona como única fonte de verdade do sistema. Ambas aplicações foram implementadas em Python e foi criada uma banca de testes compostas de três Raspberry Pis representando as diferentes camadas. Como orquestrador, foi utilizado o k3s, uma distribuição do orquestrador Kubernetes. A comunicação entre Agentes e Servidor é realizada utilizando chamadas remotas de função implementadas com o *framework* gRPC e as informações enviadas são de consumo de energia, vindos de um sistema de gerenciamento de bateria (BMS) acoplado aos *Raspberry Pis*.

RESULTADOS E DISCUSSÃO

O sistema passou por uma avaliação de performance sobre o consumo de memória volátil e consumo de CPU. O consumo de memória volátil foi significamente maior do que simplesmente rodar a aplicação sem nenhuma sobrecarga local nos nós devido as aplicações dos orquestradores, ainda que o consumo seja negligível comparável com a capacidade total de memória do sistema.

Dois casos de uso da arquitetura e da implementação foram desenvolvidos. O primeiro caso de uso explora a troca de uma aplicação de um nó de baixa capacidade computacional alimentado por bateria para (nó em Camada de Borda) para um nó de alta capacidade computacional (na Camada Cloud Pública ou de Borda). O segundo caso de uso explora a implementação de um algoritmo de classificação de melhor nó que leva em consideração o contexto e parâmetros do sistema distribuído.

CONSIDERAÇÕES FINAIS

Trabalhos futuros podem explorar algumas áreas como uso de *remoteproc* ou *RPGms* para reprogramação e interação direta entre dispositivos da Camadade Borda e da Camada de Sensores Inteligentes, adicionar período variável de amostragem das métricas de sistema, utilizar softwares já estabelecidos de comunicação como *fluentbit* ao invés do framework de monitoramento desenvolvido no trabalho, integrar ferramentas de integração contínua/entrega contínua para updates automáticos em dispositivos IoT e realizar estudos sobre a modelagem de consumo de diferentes orquestradores.

Palavras-chaves: Virtualização. Internet das Coisas. Sistemas Embarcados.

ABSTRACT

A striking feature of embedded system projects in recent years is the adoption of cloud technologies such as containerization and use of scalable cloud computing services in order to manage fleets of Internet of Things devices. This work proposes to continue the trend of using cloud-originated technologies in embedded systems with the study and implementation of a fleet management architecture using Kubernetes. The work provides a historical recap of the use and development of lightweight virtualization on systems running Linux, and proposes and implements a software architecture using an orchestrator, supported by use cases with the goal of being a first step for distributed Internet of Things projects.

Keywords: Virtualization. Internet of Things. Embedded Systems.

LIST OF FIGURES

Figure 1 – Simplified schematic of the MMU 19
Figure 2 – Architectural view of virtual machines running on infrastructure virtualized by the hypervisor. 20
Figure 3 – Architectural view of containers running on top of an operating system using a container engine as interface. 21
Figure 4 – General architecture highlighting the decoupling generated by the use of the OCI runtime. 24
Figure 5 – Container life cycle. 25
Figure 6 – Simplified Kubernetes Architecture. 26
Figure 7 – Container runtime architecture using Kubernetes. 28
Figure 8 – High-level diagram illustrating the concept architecture. 31
Figure 9 – Expanded view of the proposed architecture highlighting its technological components. 33
Figure 10 – Database table used to store the information coming from the layers. . . 34
Figure 11 – Mixed UML diagram representing the structure and basic business logic of the monitoring applications. 36
Figure 12 – Flowchart illustrating how the information is gathered between the server and client. 38
Figure 13 – Topology of the physical system. 40
Figure 14 – Single-server Setup with an Embedded Database. 42
Figure 15 – Volatile Memory consumption over time. 42
Figure 16 – Volatile Memory consumption over time. 43
Figure 17 – Edge Node has a running Pod and has 100% battery capacity. 44
Figure 18 – Node falls to 20% battery, orchestrator automatically detects this through the monitoring framework and creates a Pod on the Public Cloud Layer. 45
Figure 19 – A service watching the database, with battery information fed by the monitoring framework, switches the Pod to run in the Cloud, letting the Node recharge its battery or be serviced. 45

LIST OF TABLES

Table 1 – Namespaces on the Linux Kernel and what each isolates. 23

LISTINGS

Listing 1 – Example of a Kubernetes data description, in this case, a deployment of the nginx web server. 28
Listing 2 – Memory and CPU usage of a nginx service running inside a Docker container, not considering the orchestrator overhead. 34
Listing 3 – Opening a communication channel on port 50051 of a *monitoring-agent*. 37
Listing 4 – Creating a *stub* from the channel to call remote procedures on. 37
Listing 5 – Attributing the value of the remote call on the stub to `battery_current_response`. 37
Listing 6 – Result of the remote procedure call from on the stub from Listing 5. . . 37
Listing 7 – Insertion SQL command and custom `execute` to enter data into the Postgresql database. 38
Listing 8 – Dockerfile for the monitoring applications based on the Alpine Linux Distribution. 39
Listing 9 – Cap. 39
Listing 10 – Memory and CPU usage of a nginx service running inside a Docker container, not considering the orchestrator overhead. 42
Listing 11 – Example of a Kubernetes data description, in this case, a deployment of the nginx web server. 45
Listing 12 – Online methods to calculate the mean and variance of a given set. . . . 47
Listing 13 – Method to establish the best node giving a wide range of parameters. . 47
Listing 14 – Code for the *monitoring-server* portion of the framework seen in Figure 9. 53
Listing 15 – Code for the *monitoring-agent* portion of the framework seen in Figure 9. 55
Listing 16 – Database class used to create a database, connect, execute commands, queries and table creation and destruction on a PostgreSQL database instance. 58
Listing 17 – `requirements.txt` file, which contains the dependencies for the monitoring python applications. 61
Listing 18 – Example of context-reactive Pod switch using the database and Kubernetes APIs. 61
Listing 19 – RunningStatistics class. 62
Listing 20 – Complete example using a custom node ranking algorithm using information from the database of the monitoring application and the Kubernetes API. 64
Listing 21 – YAML file Kubernetes deployment description of the *monitoring-agent* application using *Daemonset* to deploy an agent on every node in the cluster. 66
Listing 22 – YAML file Kubernetes deployment description of the *monitoring-server* application. 67

Listing 23 – YAML file describing a virtual cluster using *kind* with two worker nodes and one control plane node. 68
Listing 24 – Script to start the *kind* virtual cluster. 69
Listing 25 – Bash script used to get data used to generate Figure 15. 70
Listing 26 – Perl script used to get data used to generate Figure 16. 70
Listing 27 – Pulling a ubuntu image from the default DockerHub registry using the Docker CLI. 71
Listing 28 – Files that compose a container image. 71
Listing 29 – Excerpt from the metadata file highlighting the Cmd section. 72
Listing 30 – A primer on using Docker and Docker Compose. 72
Listing 31 – Namespaces created when running *bash* inside a Debian container . . . 72
Listing 32 – Changing the default command on the upstream Ubuntu container image by using a Dockerfile. 73
Listing 33 – Using the Docker CLI `build` command to build a container image from a Dockerfile description. 73
Listing 34 – Running the newly built container with the modified command. 73
Listing 35 – An YAML file to be used with the Docker Compose tool. 74
Listing 36 – Result of running `docker compose up` with the YAML file listed at Listing 35 as input. 74

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
BMS	Battery Management System
BSD	Berkeley Software Distribution
CI	Continuous Integration
CPU	Central Processing Unit
CRI	Container Runtime Interface
DHCP	Dynamic Host Configuration Protocol
IoT	Internet of Things
ISC	Internet Systems Consortium
IT	Information Technology
JSON	JavaScript Object Notation
LXC	Linux Containers
MAC	Media Access Control Address
MMU	Memory Management Unit
MQTT	MQ Telemetry Transport
OCI	Open Container Initiative
OTA	Over The Air
OT	Operational Technology
PaaS	Platform as a Service
REST	Representational State Transfer
SBC	Single Board Computer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine
YAML	YAML Ain't Markup Language

CONTENTS

1 INTRODUCTION . . . 18
1.1 OBJECTIVES . . . 18

2 BACKGROUND ON VIRTUALIZATION AND CONTAINERS . 19
2.1 VIRTUALIZATION . . . 19
2.2 LIGHTWEIGHT VIRTUALIZATION AND CONTAINERS . . . 20
2.3 HISTORICAL DEVELOPMENT OF SUPPORT FOR CONTAINERS IN THE LINUX KERNEL . . . 22
2.4 ORCHESTRATORS . . . 25
2.4.1 Kubernetes Architecture . . . 26
2.5 STATE OF THE ART OF CONTAINERS AND ORCHESTRATORS IN EMBEDDED SYSTEMS . . . 28

3 DEVELOPMENT OF A MONITORING FRAMEWORK: BRIDGING IOT AND KUBERNETES . . . 30
3.1 ARCHITECTURE DESCRIPTION . . . 30
3.2 IMPLEMENTING THE MONITORING . . . 32
3.3 DATABASE SCHEMA . . . 33
3.4 DEFINITION OF COMMUNICATION BETWEEN AGENTS AND SERVER 34
3.5 STRUCTURE OF AGENT AND SERVER APPLICATIONS . . . 36
3.6 COMPLETE INFORMATION FLOW EXAMPLE . . . 37
3.7 PACKAGING OF APPLICATIONS INTO CONTAINERS . . . 39
3.8 PHYSICAL CLUSTER . . . 40
3.8.1 Hardware and Networking . . . 40
3.9 SOFTWARE SETUP . . . 41
3.9.1 Performance considerations and benchmarking . . . 41
3.10 CHANGING THE CLUSTER STATE BASED ON CONTEXT INFORMATION . . . 43
3.11 USE-CASES FOR THE MONITORING FRAMEWORK . . . 44
3.11.1 Use-case number 1 for the Monitoring framework: Context-Reactive Pod Switch . . . 44
3.11.2 Use-case number 2 for the Monitoring framework: Context-Aware Scheduling . . . 46

4 CONCLUSION . . . 48
4.1 DISCUSSION . . . 48
4.2 FUTURE WORKS . . . 48

REFERENCES 49

APPENDIX A – MONITORING FRAMEWORK CODE AND DEPLOYMENT FILES 53

A.1 FRAMEWORK CODE 53

A.2 DEPLOYMENT OF THE MONITORING FRAMEWORK ON KUBERNETES 66

A.3 SIMULATING THE CLUSTER WITH *KIND* 68

APPENDIX B – SCRIPTS USED TO GATHER RUNTIME PARAMETERS 70

APPENDIX C – A PRIMER ON USING DOCKER AND DOCKER COMPOSE. 71

C.1 DOCKER 71

C.2 DOCKER COMPOSE 74

1 INTRODUCTION

The Internet of Things (IoT) is an abstraction for wirelessly connected devices that also integrates with a cloud component (ATZORI; IERA; MORABITO, 2010). Although on its infancy there are more than 13 billion IoT devices deployed in 2022, which is to almost triple from 9.7 billion in 2020 to more than 29 billion in 2030 (STATISTA, 2022).

Some Industrial IoT (IIoT) applications are designed to run in a distributed manner, with needs of highly responsive (ie, low-latency) and/or physically close systems, giving name to the *edge-native* paradigm (CHIMA OGBUACHI et al., 2020). Several technologies from the so-called *cloud-native* sphere such as containerization and orchestration are being actively used to deploy distributed IoT applications, manage fleets and integrate these devices with powerful cloud networks, quite literally entering consumer's homes everyday with smart home devices, for example.

The commoditization of cloud services with massive, on-demand computing offerings is driven by technologies such as container orchestration, which offer scalability and high availability (i.e., no downtime) when deploying software services, enabling more distributed architectures, such as microservices. In this sense, there is a strong argument for the use of not only lightweight virtualization technologies but also the use of orchestrators in Internet of Things systems.

One of the issues is that the value of variables of interest within a distributed IoT system necessary to properly monitor and schedule loads in the system are not currently taken into account within modern orchestrator frameworks (CHIMA OGBUACHI et al., 2020). Thus, in order to continue the trend of using cloud technologies to manage distributed embedded systems.

1.1 OBJECTIVES

This dissertation aims to provide an initial starting point for Internet of Things projects that wish to use an orchestrator as a fleet management and monitoring system, creating the necessary applications to amend the aforementioned limitation of current orchestrators.

As specific objectives we have:

- Provide a general overview of the historical developments within virtualization and orchestrators;
- Describe the proposed reference architecture for distributed IIoT applications using virtualization techniques;
- Describe the development of a monitoring framework for distributed IIoT applications based on the proposed architecture;
- Benchmark and present use-cases cases for the monitoring framework.

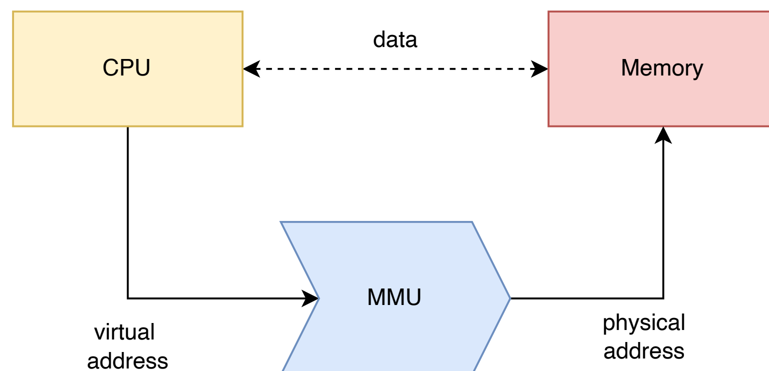
2 BACKGROUND ON VIRTUALIZATION AND CONTAINERS

2.1 VIRTUALIZATION

According to (BUGNION; NIEH; TSAFRIR, 2017), "Virtualization is the application of the layering principle through enforced modularity, whereby the exposed virtual resource is identical to the underlying physical resource being virtualized".

The canonical example of Virtualization is the Memory Management Unit (MMU), which translates virtual addresses to physical ones (BUGNION; NIEH; TSAFRIR, 2017), enabling mechanisms such as memory protection between processes. Figure 1 represents how the MMU acts as a system module exposing the physical memory resource to the Central Processing Unit (CPU).

Figure 1 – Simplified schematic of the MMU.



Source: by the Author.

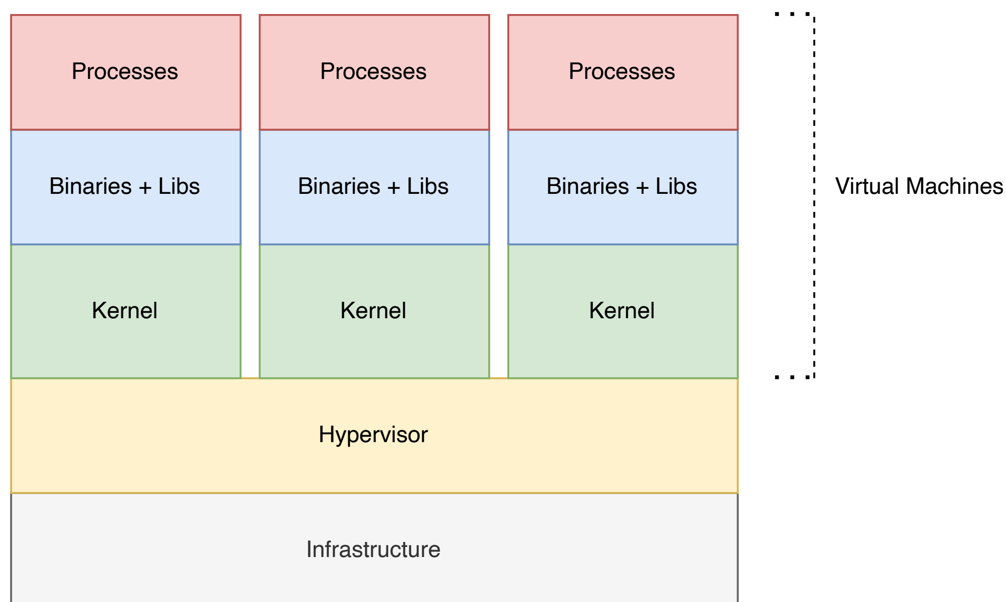
An example on the software side there is the Virtual Machine, which according to (BUGNION; NIEH; TSAFRIR, 2017) is "[...] a complete compute environment with its own isolated processing capabilities, memory, and communication channels."

The defining feature of the Virtual Machine is to execute on top of a software layer while providing the features and functionalities as if it was executing on of real hardware. One of the mechanisms used to achieve this feat is a software layer named *hypervisor*, also called a *Virtual Machine Monitor* (POPEK; GOLDBERG, 1974). The function of the hypervisor is to virtualize the hardware for virtual machines running each their own operating system and applications.

A complete treaty of Virtual Machines is out of scope for the present study, but it

is necessary to understand that virtual machines[1] will apply the virtualization principle to all components of the system, each having a virtualized copy of the underlying hardware (BUGNION; NIEH; TSAFRIR, 2017), meaning a completely new and separate kernel used for each virtual machine. This is highly desirable due to the higher level of isolation between the virtualized and physical system, while also having common drawbacks of traditional bare-metal systems (longer boot times, harder-to-use version control systems with it, and difficulty in scaling).

Figure 2 – Architectural view of virtual machines running on infrastructure virtualized by the hypervisor.



Source: by the Author.

2.2 LIGHTWEIGHT VIRTUALIZATION AND CONTAINERS

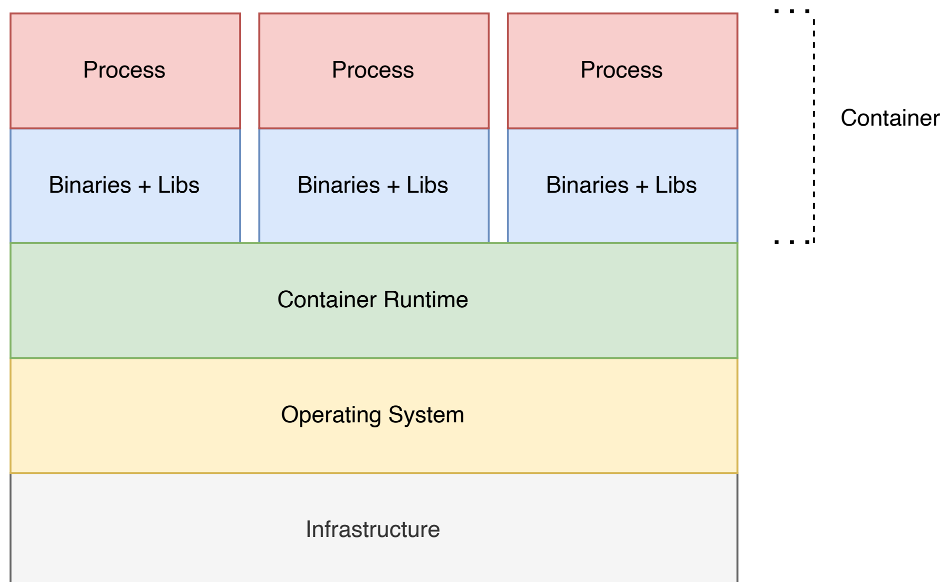
The aforementioned drawbacks of the virtual machines is lessened with another type of virtualization technique, namely *lightweight virtualization* (or its less academic denomination, *containers*), which according to (BUGNION; NIEH; TSAFRIR, 2017) "[...] rely on a combination of hardware and software isolation mechanisms to ensure that applications running directly on the processor ... are securely isolated from other sandboxes and the underlying operating system."

Instead of using a Hypervisor and running a new operating system, containers rely on their host operating system to create a virtualization layer, commonly referred to as *container*

[1] Here the word "virtual machine" is used in the academic sense of a *system-level virtual machine*. Information regarding semantics can be found in (BUGNION; NIEH; TSAFRIR, 2017), albeit the industry largely ignores the other definitions for virtual machines.

runtime, by enforced modularity (BUGNION; NIEH; TSAFRIR, 2017)[2]. This architectural difference is highlighted between Figure 2 and Figure 3. Not using a hypervisor with a guest operating system makes the raw performance and boot times of containers faster than VMs with Linux-based Hypervisors, although at the cost of applying and managing security, users and network policies (RAD; BHATTI; AHMADI, 2017).

Figure 3 – Architectural view of containers running on top of an operating system using a container engine as interface.



Source: by the Author.

Containers became rapidly popular and supported by the main cloud computing platforms since the technology addresses some of the issues (CASALICCHIO, 2019) faced by deploying applications on the cloud such as:

- *Dependency hell*, where different applications may depend on different versions of libraries and need to coexist inside the same server: containers solve this by having separate binaries and libraries, and as we shall see are much easier to maintain than virtual machines.
- *Application portability*, where a container can be executed independent of anything other than a compatible container runtime (see figure Figure 3 and section 2.3, which provides a background on the historical aspects of standardization, which provides much of the application portability).
- *Performance overhead problem*, with no need of a hypervisor and having its own kernel per instance (as opposed to VMs), containers show a negligible overhead (MORABITO;

[2] Some Hypervisors (namely, Type-2 Hypervisors like VirtualBox™and VMWare Workstation™) also rely on some layer running inside along an operating system

KJÄLLMAN; KOMU, 2015) and are an order of magnitude faster to stop, start and restart an instance (CASALICCHIO, 2019). Although much of the literature shows performance comparisons between the VM and Container approaches, this is in practice shadowed by the architectural needs of a given project.

2.3 HISTORICAL DEVELOPMENT OF SUPPORT FOR CONTAINERS IN THE LINUX KERNEL

In 1979, with the release of the Unix V7, the `chroot` (short for *change root*) system call was introduced (LABORATORIES, 1979), which sets the *root* of the Unix file system to a given directory. This means different processes can have completely segregated file systems. The system call is present to this day on every major **nix*-like system(such as BSD and Linux).

In 2007, engineer Paul Menage from Google introduced a series of patches into the Linux Kernel that created what is now called `cgroups`, or control groups, which are "[...] a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored". (CGROUPS(7). . . , 0021).[3]

The capabilities of `cgroups` are (CGROUPS(7). . . , 0021):

- Limiting Resources such as memory, file system cache, I/O bandwidth, and CPU limits.
- Prioritization of groups for CPU utilization and disk I/O.
- Metrics for group resource usage.
- Freezing and restarting groups of processes.

Another major component that makes it possible for Lightweight Virtualization is namespaces. A namespace "[...] wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource" (NAMESPACES(7). . . , 0021). Table 1 lists the existing namespaces currently supported in the Linux Kernel.

In 2008, a tool called *LXC*, short for Linux Containers, was released to the public called LXC. LXC makes use of the *cgroups* and *namespaces* features of the Linux Kernel making it easy to set up containers. LXC is thought of as "...something in the middle between a chroot and a full-fledged virtual machine. The goal of LXC is to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel." (LXC, n.d.) LXC is focused on virtualizing stateful Linux Operating Systems inside Linux.

[3] Curiously, the name "containers" was already too overloaded by then, and in the patch e-mail itself Paul asks for the opinion of Linus " - decide whether "Containers" is an acceptable name for the system given its usage by some other development groups, or whether something else (ProcessSets? ResourceGroups? TaskGroups?) would be better. I'm inclined to leave this political decision to Andrew/Linus once they're happy with the technical aspects of the patches."

Table 1 – Namespaces on the Linux Kernel and what each isolate.

Namespaces	Flag	Isolates
Cgroup	CLONE_NEWCGROUP	Cgroup root directory
IPC	CLONE_NEWIPC	System V IPC, Posix message queues
Network	CLONE_NEWNET	Network devices, stacks, ports etc
Mount	CLONE_NEWNS	Mount points
PID	CLONE_NEWPID	Process IDs
Time	CLONE_NEWTIME	Boot and monotonic clocks
User	CLONE_NEWUSER	User and group IDs
UTS	CLONE_NEWUTS	Hostname and NIS domain

Source: (NAMESPACES(7)..., 0021).

In 2013, Google released the source code for its container tool *lmctfy*, an acronym for "Let Me Contain That For You" (GOOGLE, 2013). Although LXC and *lmctfy* are very similar and work on the same level, they had some fundamental differences that would shape the future of the Linux Containers scene.

According to Tim Hockin, one of the contributors to *lmctfy* and who would later be a co-founder of the Kubernetes Projects, "We took a different approach and made it just a bit more abstract, so users don't have to understand cgroups so much" (HOCKIN, 2013). The choice of higher levels of abstraction with a focus on ergonomics and ease of use steered much of the containerization technology in Linux for the next years.

Also in 2013, a tool called Docker was released as open-source by dotCloud, a company that focused on Platform as a Service (PaaS). The main difference between Docker and the other tools was the level of abstraction, functionality, and ergonomics given to the user. Docker built upon established technologies such as LXC, using it as its Linux Container runtime tool before Docker version 0.9 and became the *de facto* container management tool for containerization, and the first commercially successful solution(RAD; BHATTI; AHMADI, 2017). Readers not familiar with using Docker should refer to Appendix C.

As companies started using Docker as part of their core tooling, the necessity grew for standardization of the myriad of different software modules. Thus, the Linux Foundation created the Open Container Initiative, which aims to establish open standards for interoperability between different tools in the container ecosystem.

In 2015, the *lmctfy* project would stop and the efforts were geared towards developing *libcontainer*, which was proposed as a replacement for *lmctfy* (JNAGAL, 2015), but ended up replacing LXC as the container runtime for Docker, renamed to *runc* and donated to the Open Container Initiative (OCI) as its first project.

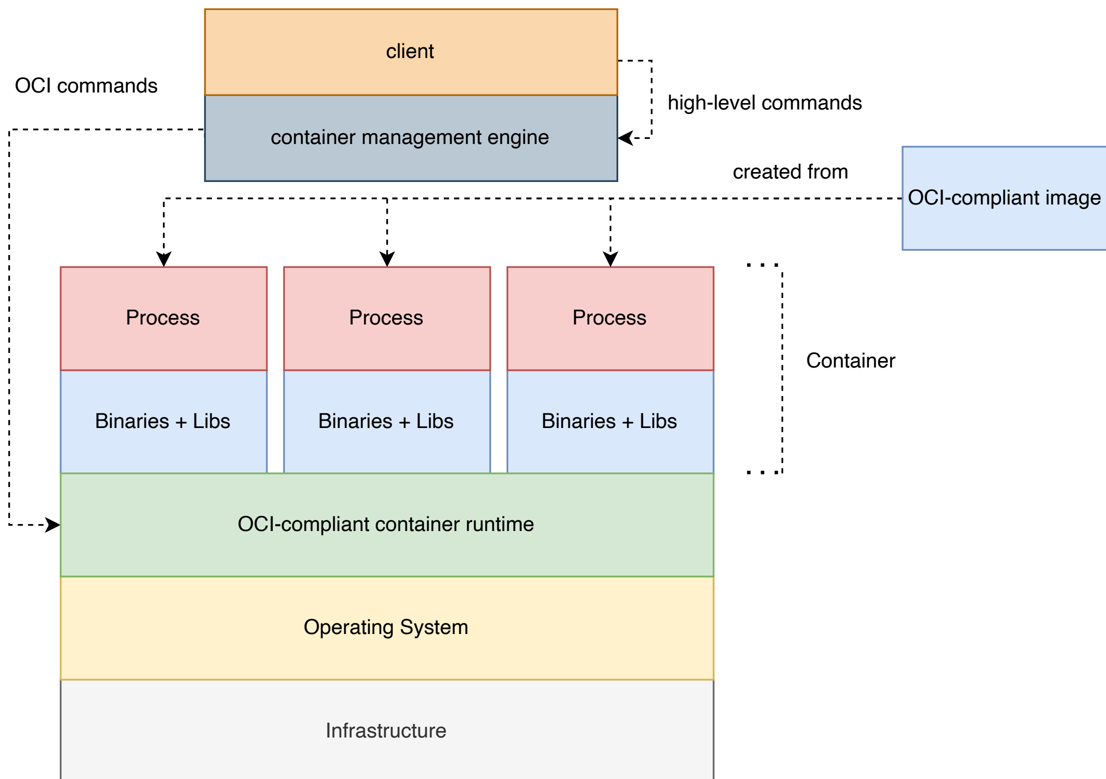
The two main standards provided by the OCI are the Runtime Specification and the Image Specification.

The Runtime Specification defines the interface between the container runtime and the operating system kernel. It specifies a set of common operations that should be supported by

the container runtime such as *create*, *query*, *start*, *kill*, and *delete*, as well as the lifecycle of a container (OCI, 2015).

The Image Specification defines the format, environment variables, and storage and retrieval mechanisms of a container image, which are files containing everything needed to run a container, such as application code, libraries, and metadata (OCI, 2015).

Figure 4 – General architecture highlighting the decoupling generated by the use of the OCI runtime.



Source: by the Author.

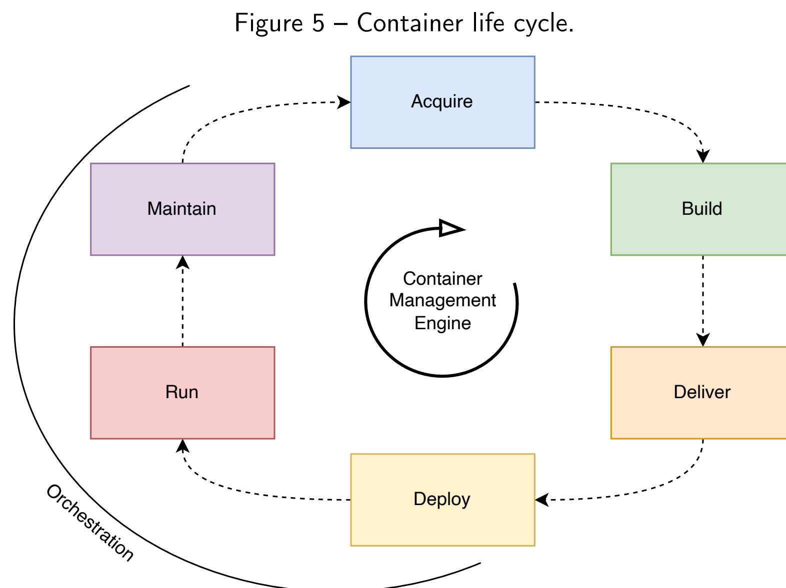
Figure 4 exemplifies how the modules are integrated within an OCI runtime, including the OCI-compliant runtime, and how containers are created from the OCI-compliant image format. Note that instead of "Linux Kernel" the figure simply mentions "Operating System", which can in effect be any operating system that has container capabilities, so although much of the discussion was centered around Linux, the scope for interoperability between operating systems, container management engines, clients and infrastructure is much bigger than that. The essential takeaway is that containers, via the decoupled architecture provided by the OCI standardization efforts, can be manipulated by a myriad of tools, such as orchestrators, as we shall see (subsection 2.4.1).

2.4 ORCHESTRATORS

Containers[4] are designed to run one application at a time and are generally built using a series of OCI-image layers (see Appendix C for a practical example using Docker), ending up with a multi-layer OCI image, which is stateless, eg, the contents written on this last layer are lost after the container stops (CASALICCHIO, 2019).

Containers are thus subject to a life cycle, as shown in Figure 5, which is managed by the container runtime engine, and can be broken down into select states: *acquire*, starting from a trusted base image; *build*, layering your application and libraries on top of the base image; *deliver*, bringing the now complete image to an artifacts storage system; *deploy*, the actual step of updating the application into production; *run*, determining how to schedule your application to execute, as well as setting up network connections, fallbacks and scaling policies; *maintain*, constant supervision of application status, automatic recover and fallback

Figure 5 – Container life cycle.



Source: adapted from (CASALICCHIO, 2019).

Issues arise when the *deployment*, *running*, and *maintaining* cycles are considered in the context of scale: deploying hundreds or thousands of containers is out of scope for the container management engine, which cannot effectively handle issues such as resource limit control, scheduling, load balancing, health checks, fault tolerance and autoscaling (CASALICCHIO, 2019). A higher abstraction layer controlling one or many container management engines is needed, which is where the container orchestrator comes in.

According to (CASALICCHIO, 2019), "[...] Container orchestration allows cloud and application providers to define how to select, to deploy, to monitor, and to dynamically control

[4] The word "container", unless specifically mentioned, shall be used exclusively for *application containers* from now onwards. For more information on the taxonomy of container technologies, please refer to (CASALICCHIO, 2019)

the configuration of multi-container packaged applications in the cloud". Several container orchestration solutions have been developed in the last decade, examples including Docker Swarm, Kubernetes, Apache Mesos, and Nomad.

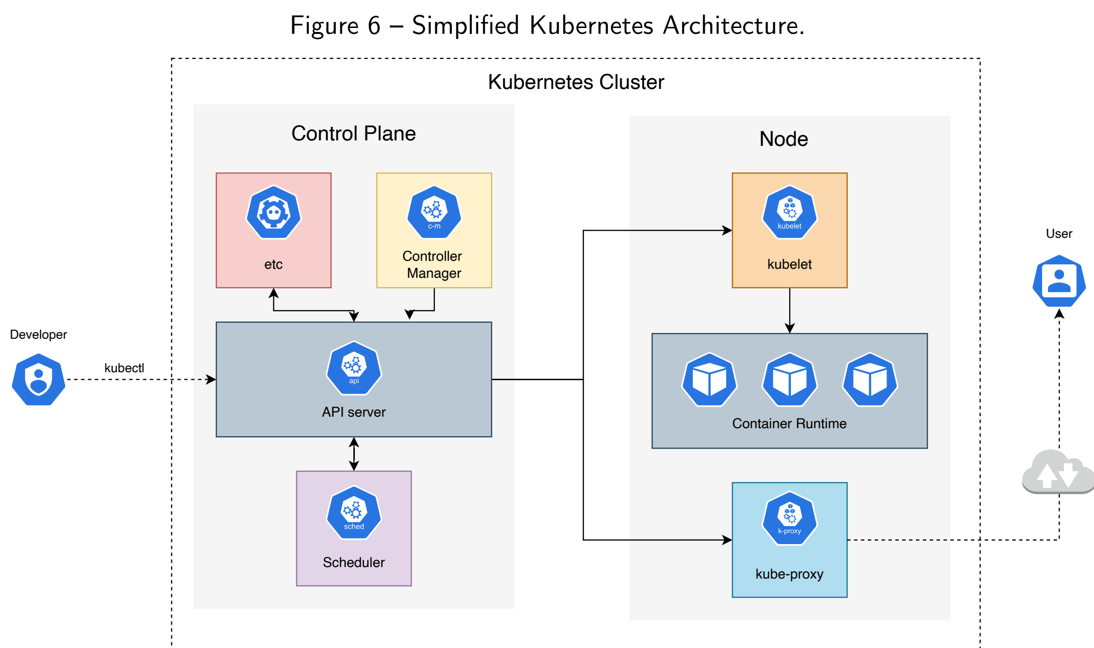
Kubernetes is particularly interesting given the fact it is the result of decades of experience deploying thousands of containers at Google, released as open source with the intent of "[...] making it easy to deploy and manage complex distributed systems" (BURNS et al., 2016). Kubernetes was also the seed project for the Cloud Native Computing Foundation, in partnership with the Linux Foundation. 96% of respondents of the 2021 Cloud Native Computing Foundation have answered to be either evaluating or already using Kubernetes (CNCF, 2021). If Docker is the *de facto* container management engine, so is Kubernetes for orchestrators.

2.4.1 Kubernetes Architecture

The main components of Kubernetes are the nodes and the control plane, which is a special node. Nodes can be physical or virtual machines that have a container runtime. Each node runs an atomic entity denominated Pod, which is a collection of containers.

Within each node, there are three main components: the kubelet, the *kube-proxy*, and the container runtime. The control plane is composed of the API server, a key-value store (*etcd*, but in some cases this can be swapped by other data storage systems, such as SQLite, in the case of the k3s distribution), the *controller manager* and the *scheduler*. Figure 6 shows the internal components, with a short description of each following.

Figure 6 – Simplified Kubernetes Architecture.



Source: adapted from Ashish Patel.

The API server is a RESTful API for interacting with the Kubernetes cluster, designed to provide a frontend to the cluster's shared state. It allows updates, deletions, and updates

to Kubernetes objects, either via a programming language client library that implements the necessary HTTP requests or using the *kubectl* command line tool.

The *etcd* is a key-value storage solution that works as the single source of truth for the cluster. It's a stateful component for all cluster configurations and the current state.

The scheduler is responsible for scheduling Pods with no assigned nodes to be deployed on available nodes. Some factors taken into account for node selection: include resource requirements and several constraints such as node affinity and anti-affinity, selections based on labels, taints, and tolerations.

Controllers in Kubernetes continuously monitor the state of the cluster through the API server and make the necessary adjustments to move the cluster to the desired state, akin to control loops from control theory. As there are many controllers inside Kubernetes (node, jobs, endpoints, and service controllers to name a few), they are co-located inside the functional unit of the controller manager.

The kubelet runs in every node composing the Kubernetes cluster. Initially, the kubelet communicates with the API server to register to the Control Plane the properties and identification of the node that is executing that particular kubelet instance, subsequently requesting the desired state for that particular node and taking the necessary steps to achieve that state. The kubelet works in terms of a *PodSpec*, which is a YAML or JSON object that describes a Pod, ensuring that the Pods described by the PodSpec are running healthily.

The kube-proxy is one of the core entities in the Kubernetes networking model, allowing communication between inside and outside the cluster, ensuring each Pods gets a unique IP and that containers running inside the same Pod get the same IP, massively reducing the network complexity.

The Container runtime, as seen in section 2.2, is the component that will effectively run the containers, and Kubernetes supports a variety. In effect, any container runtime that implements the Kubernetes Container Runtime Interface (CRI) can be used. The Container Runtime Interface was specifically created to enable a decoupled architecture between the kubelet and the container runtime. The CRI was born out of necessity to stop using software "shims" between the container management engine and the container runtime: container runtimes themselves can instead implement the CRI. Figure 7 represents a Kubernetes specific stack of Figure 3, highlighting how the kubelet, kube-proxy, and CRI are connected to the API server and the rest of the infrastructure.

A sometimes confusing aspect of Kubernetes is that the components that manage Kubernetes (for example, the *kubelet*, *etcd* or the API server) are themselves deployed as Kubernetes pods.

Kubernetes is deliberately declarative, with a strong separation between computation and data (BURNS et al., 2016). The data representing the desired state of the cluster is applied to the configuration using a high-level language such as YAML. The code excerpt in Listing 1 shows the description of an nginx web server deployment. This file is applied by the developer using the *kubectl* tool as in Figure 6.

Figure 7 – Container runtime architecture using Kubernetes.

Source: by the Author.

Listing 1 – Example of a Kubernetes data description, in this case, a deployment of the nginx web server.

```
1
2    apiVersion: v1
3    kind: Pod
4    metadata:
5    name: nginx
6    spec:
7    containers:
8    - name: nginx
9    image: nginx:1.14.2
10   ports:
11   - containerPort: 80
```

2.5 STATE OF THE ART OF CONTAINERS AND ORCHESTRATORS IN EMBEDDED SYSTEMS

The use of container technologies in Embedded systems is new but mature. Previous works such as (LAMMI, 2018) and (NORONHA et al., 2018) have shown a possibility of using in containers with Embedded systems. Commercial offerings such as Foundries.io™, balenaOS™ and Torizon™ make use of containers in embedded devices to accelerate customer's *time-to-market*.

The review done for this chapter revealed no commercial interest in supporting container orchestration for IoT devices on the edge, and this may be due to the tooling limitations as described in (LEE et al., 2021) and (CHIMA OGBUACHI et al., 2020), the latter which

specially highlights the lack of context information, ie, the various parameters influencing an IoT deployment. The subject of chapter 3 is thus the development of a reference, generic software architecture and a possible implementation of a framework to enable context-aware monitoring of container orchestrators in embedded systems.

3 DEVELOPMENT OF A MONITORING FRAMEWORK: BRIDGING IOT AND KUBERNETES

In this chapter, we present an attempt of bridging IoT and orchestrator frameworks such as Kubernetes. One of the main challenges is adjusting the orchestrator software to take in consideration the many parameters that have effect in an IoT product application, and for that we develop our own monitoring solution from a reference architecture, jump starting possible product developments that may benefit from such a distributed system.

The chapter is organized as follows:

1. Architecture Description: propose a software architecture for a orchestration system that defines the communication and data flow from a number of heterogeneous layers to a single orchestration system.

2. Implementing the Monitoring Framework: implement the architecture described on the previous chapter using the Python language and a database as a single source-of-truth for the system.

3. Physical Cluster: configure a physical hardware test-bench using Raspberry Pis as a demonstrator for the Monitoring Framework and architecture.

4. Software Setup: describes how a Kubernetes distribution was used to setup the "orchestrator" portion of the architecture. Also contains some performance and benchmarking tests.

5. Changing the Cluster State Based on Context Information: describes how, based on the data gathered in the database using the monitoring framework, it is possible to change cluster properties using the Kubernetes API.

6. Use-cases for the Monitoring Framework: describes two use cases using the control-loop derived from the previous chapters using the monitoring framework and the Kubernetes API.

3.1 ARCHITECTURE DESCRIPTION

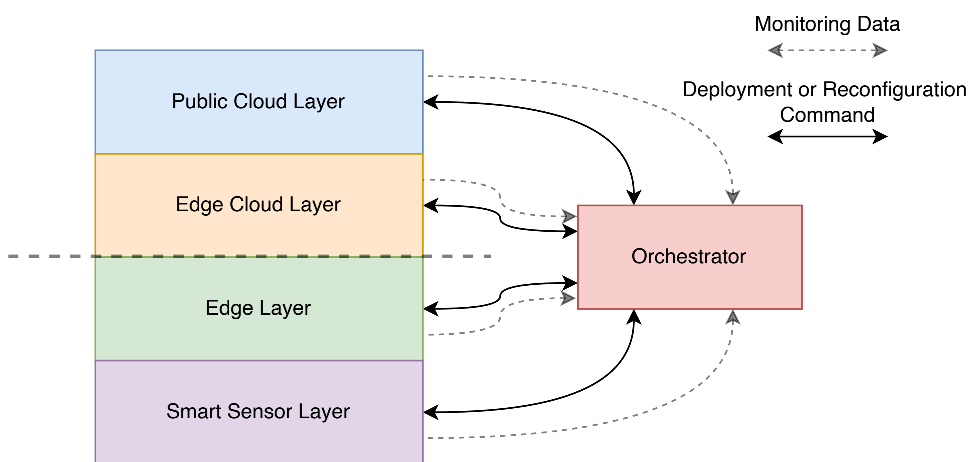
The Architecture is such that the Operational Technology (OT) layer can interact with the Information Technology (IT) layer via the Orchestrator (Figure 8). The OT and IT levels are stacked such that the computational power (and hence the energy consumption) increase going from top to bottom.

The OT Level will contain Edge and Smart Sensor devices. Edge devices in this context are devices running embedded Linux, with low to medium computational capacity and the ability to run Linux Containers. Smart Sensor Layers will generally be Microcontrollers or Programmable Logic Controllers connected to the system via the MQTT protocol.

The IT Level contains the Edge Cloud and the Public Cloud. The main difference between both is latency: the Edge Cloud acts as a low-latency, local infrastructure, while the Public Cloud is an on-demand computational instance from a Cloud Service Provider such as Amazon Web Services and Google Cloud Platform. The Edge Cloud can be seem as the edge of the public internet infrastructure (the Public Cloud).

Each computational entity inside of each layer is hereafter named "node". On the Edge Layer we may have several Nodes composed each of Raspberry Pis and other Linux-capable devices, on the Edge Cloud we may have several instances of Virtual Machines, each instance being a Node, and on the Smart Sensor Layer we may have several Microcontrollers, also individually abstracted as Nodes.

Figure 8 – High-level diagram illustrating the concept architecture.



Source: by the Author.

The Orchestrator software acts as the central point of communication and configuration for the Nodes existing inside the Layers. Two types of data exist in this Architecture:

- **Monitoring Data**, which contains information about the Nodes inside the Layers, such as battery status, energy consumption, temperature and many other metrics.
- **Reconfiguration/Deployment Commands**, which change the software currently running on each of the Nodes to complete certain functions.

Some implementation concerns arise with this architecture:

1. The Orchestrator is a communication bottleneck between the Layers, as every command has to go through what it looks like a single software service. This can lead to scalability problems if thousands of devices are expected to communicate through the same channels.
2. An interesting feature not addressed by the architecture is swapping computational loads between the layers to make it more efficient, e.g., instead of running a complex operation on an Edge Layer Node, run on the more powerful Public Cloud Layer.

3. The Monitoring Data coming from the Nodes may be too specialized to be handled by a general Orchestrator.

These can be addressed by the proper choice of the Orchestrator software, which should handle both types of communications between the layers. Kubernetes provides the first necessary communication, control and scalability facilities, as described in subsection 2.4.1. For the Monitoring Data, shortcomings were identified regarding the small set of Node information acquired by Kubernetes, lacking the necessary context of each of the Nodes (CHIMA OGBUACHI et al., 2020), composed of Node metrics (temperature, power status etc).

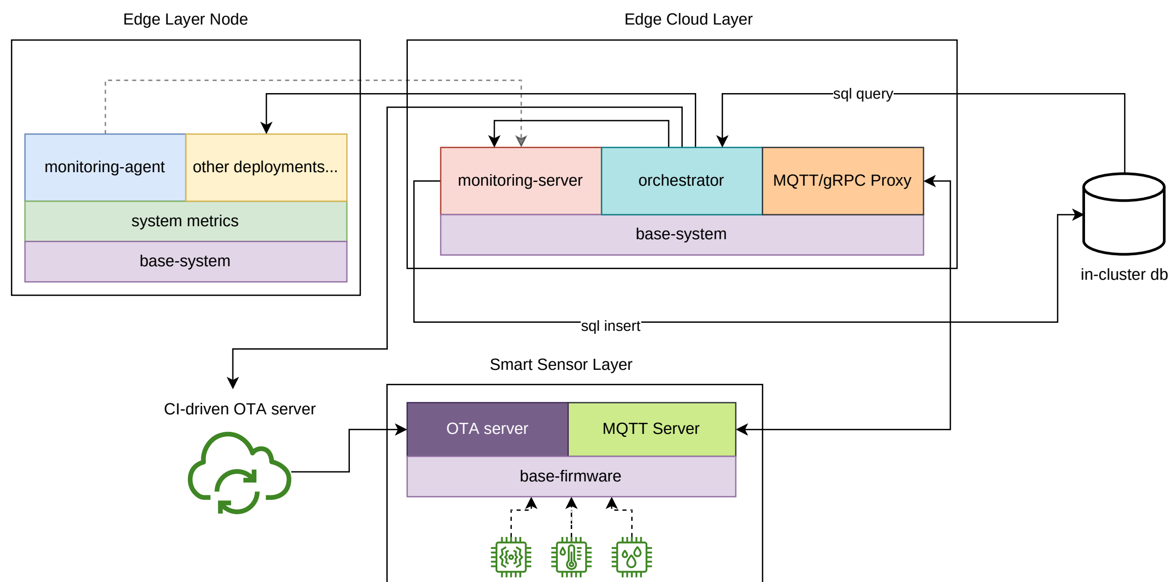
3.2 IMPLEMENTING THE MONITORING

An expanded view of the architecture is presented in Figure 9. In this view, each Layer contains a single Node, and the Public Cloud Layer is omitted. A description of each component of the deployment follows:

- **base-system**: a Linux-based system, with capabilities to run Linux Containers.
- **system-metrics**: here the system metrics are exposed to user-space as Unix files.
- **monitoring-agent**: the monitoring agent is the client-side implementation of the Monitoring framework, which reads the system metrics and send it over the network to the monitoring-server, when requested.
- **monitoring-server**: asks the information to the many monitoring agents.
- **in-cluster db**: a database to hold the Node metrics acquired via the monitoring-server, which holds historical data and decouples the queries from the main services.
- **base-firmware**: the firmware of the microcontroller. This may be bare-metal or an Real Time Operating System.
- **CI-driven OTA server**: a TCP or UDP server that runs every time a certain conditions is met at the Continuous Integration phase. The Continuous Integration agent may be managed or a cloud one (such as GitHub or GitLab).
- **OTA client**: the client for the OTA server described above. It receives the firmware from the OTA server and updates all or part of the base-firmware, altering the functions of the microcontroller.
- **MQTT broker**: an MQTT broker to enable communication between the microcontrollers and the overall system.
- **MQTT/gRPC Proxy**: to provide an opaque API for the Monitoring framework, a Proxy or a Translator is inserted into the architecture. Every call from the **monitoring-server**

is translated to an MQTT Pub/Sub routine to gather information from the Smart Sensor Layer devices.

Figure 9 – Expanded view of the proposed architecture highlighting its technological components.



Source: by the Author.

In this work, we will only be concerned with the development of the layers above the Smart Sensor Layer and will not consider the implementation of the CI-driven OTA server, which is the work of the Master's student Mohammed Noufal Ahmaed Kabir on this thesis "Realization of cloud based automated firmware OTA update of MSP over 5G", to be published.

3.3 DATABASE SCHEMA

For the database, PostgreSQL was chosen for its simplicity, ease of use and stellar documentation. A custom Python connector was written for the database, which can be examined in Listing 16 of Appendix A.

Inside the database a table *node_parameters* was created, represented in figure 10. As a part of a larger research project, this work is focused on energy consumption. The following Node metrics were chosen accordingly:

- *node_uuid*: an Universallly Unique Identifier that every node has, used to filter for individual devices.
- *battery_percentage*: the battery percentage of a node at a given time.
- *battery_voltage*: the battery voltage of a node at a given time.
- *battery_current*: the battery current at a given time.

- *gpio_voltage*: the voltage provided or received by the General Purpose Input/Output pins at a given time.
- *gpio_current*: the current provided or received by the General Purpose Input/Output pins at a given time.
- *inserted_at*: the time at which the data was inserted into the data table.

The choice of this schema already dictates some of the system's implementation, as data is captured as a packet and them inserted. The main limitation of this approach is that the period of measurements has to be the same for all node metrics.

Figure 10 – Database table used to store the information coming from the layers.

node_parameters
node_uuid
battery_percentage
battery_voltage
battery_current
gpio_voltage
gpio_current
inserted_at

Source: by the Author.

3.4 DEFINITION OF COMMUNICATION BETWEEN AGENTS AND SERVER

For the exchange of monitoring data (Figure 9) between the agents and the server, the gRPC Remote Procedure Call framework was chosen for its small message size (defined using the Protobuf data format and serialized as binaries), ideal for embedded networked applications.

The `.proto` Protobuf file (Listing 2) defines the message structure, the remote procedure calls and their respective answers. To ease implementation costs regarding sessions and node identification within the monitoring framework, we have switch the server-client model generally used with gRPC: the Nodes act as gRPC servers and the client is implemented with the **monitoring-server** service. This way, the client (monitoring-server) remotely triggers the methods with empty parameters (`google.protobuf.Empty`) and the server (monitoring-agent) returns the asked value.

Listing 2 – Memory and CPU usage of a nginx service running inside a Docker container, not considering the orchestrator overhead.

```
1     syntax = "proto3";
2
3    import "google/protobuf/empty.proto";
```

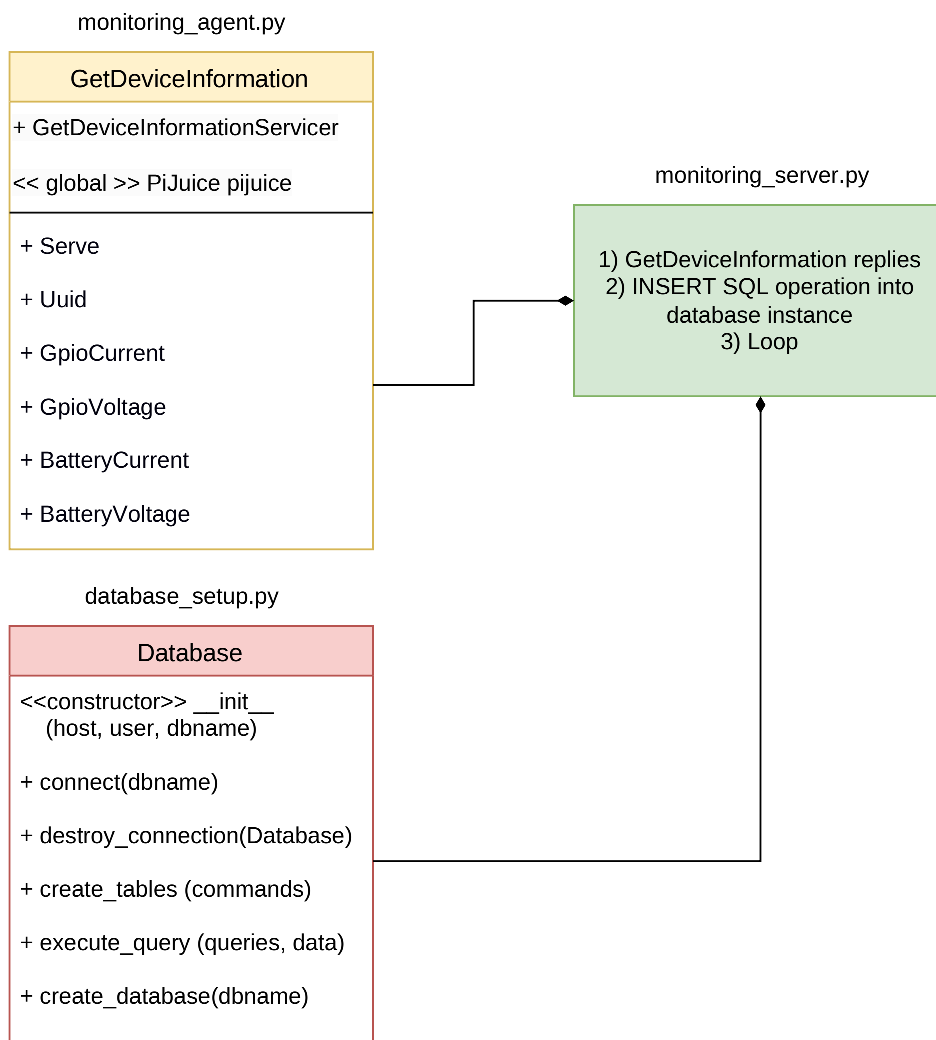
```
4
5   package deviceproperties;
6
7   service GetDeviceInformation {
8       rpc BatteryPercentage (google.protobuf.Empty) returns
           (BatteryPercentageReply) {}
9       rpc BatteryVoltage (google.protobuf.Empty) returns
           (BatteryVoltageReply) {}
10      rpc BatteryCurrent (google.protobuf.Empty) returns
           (BatteryCurrentReply) {}
11      rpc GpioVoltage (google.protobuf.Empty) returns
           (GpioVoltageReply) {}
12      rpc GpioCurrent (google.protobuf.Empty) returns
           (GpioCurrentReply) {}
13      rpc Uuid (google.protobuf.Empty) returns (UuidReply) {}
14  }
15
16  message BatteryPercentageReply {
17      int32 percentage = 1;
18  }
19
20  message BatteryVoltageReply {
21      int32 voltage = 1;
22  }
23
24  message BatteryCurrentReply {
25      int32 current = 1;
26  }
27
28  message GpioVoltageReply {
29      int32 voltage = 1;
30  }
31
32  message GpioCurrentReply {
33      int32 current = 1;
34  }
35
36  message UuidReply {
37      string uuid = 1;
38  }
```

3.5 STRUCTURE OF AGENT AND SERVER APPLICATIONS

The implementations of the `monitoring-agent` and `monitoring-server` applications were done in the Python programming language. The structure and business logic of both applications is shown in Figure 11. The monitoring server simply runs a super loop pattern, constantly feeding the database with the replies from the remote procedure calls generated by the gRPC network (section 3.4).

A necessary class is the `database`, which has all the necessary methods to connect, modify, query and update a Postgresql instance. This class is shared by all applications that need to connect to the database (section 3.11).

Figure 11 – Mixed UML diagram representing the structure and basic business logic of the monitoring applications.



Source: by the Author.

The implementation details for the agent and the server can be found in the Appendix A.

3.6 COMPLETE INFORMATION FLOW EXAMPLE

As an example of the information flow, Figure 12 shows sequential diagram, assuming that the *monitoring-server* has required the `BatteryCurrent` metric. A walk-through the code used to make this complete circuit follows:

1. The *monitoring-server* opens a communication channel within one of the **monitoring-agent** using the function call in Listing 3.

   Listing 3 – Opening a communication channel on port 50051 of a *monitoring-agent*.

   ```
   1 async with grpc.aio.insecure\_channel("monitoring-agent:50051")
         as channel:
   ```

   In Listing 3 *monitoring-agent* is automatically resolved to its corresponding IP (given by the DHCP server) using a simple DNS server also running on the network, to ease development. The corresponding port for all the Nodes is 50051.

2. The channel is used to open a *stub* (Listing 4) object that creates a remote interface to trigger remote procedure calls.

   Listing 4 – Creating a *stub* from the channel to call remote procedures on.

   ```
   1 stub =
         deviceproperties_pb2_grpc.GetDeviceInformationStub(channel)
   ```

3. The corresponding remote procedure call for the `BatteryCurrent` metric is called and its result is stored on the `battery_current_response` variable with the code in Listing 5.

   Listing 5 – Attributing the value of the remote call on the stub to battery_current_response.

   ```
   1 battery_current_response = await stub.BatteryCurrent(
   2     deviceproperties_pb2
   3     .google_dot_protobuf_dot_empty__pb2
   4     .Empty())
   ```

4. On the *monitoring-agent* side, the remote procedure call results in the `GetBatteryCurrent` method to be called (Listing 6), which sets `BatteryCurrentReply` to the value of the battery current as read from the I2C bus connecting the PiJuice to the RaspberryPi 3.

   Listing 6 – Result of the remote procedure call from on the stub from Listing 5.

   ```
   1 async def BatteryCurrent(
   2     self,
   3     request:
   4         deviceproperties_pb2.
   ```

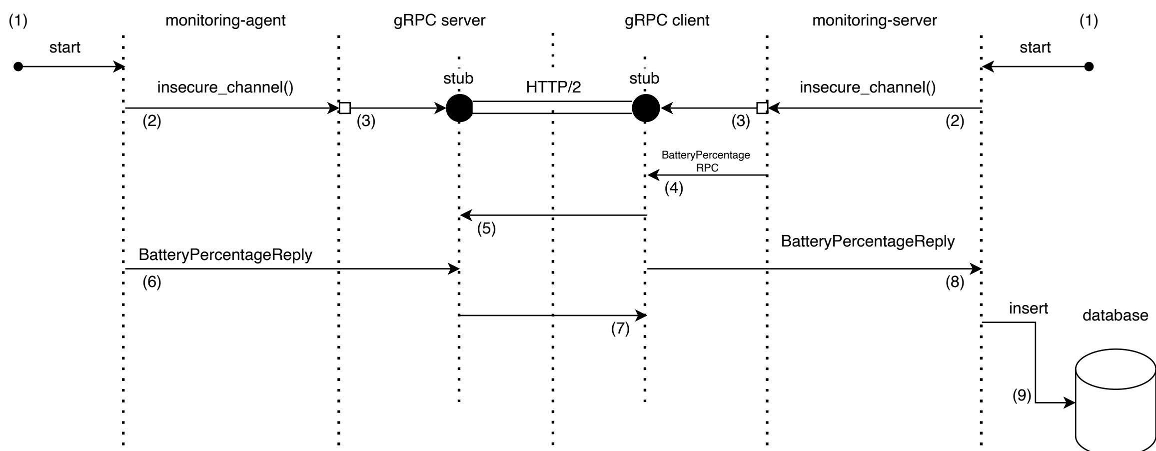
```
5             google_dot_protobuf_dot_empty__pb2.
6             Empty(),
7         context:
8             grpc.aio.ServicerContext,
9         ) -> deviceproperties_pb2.BatteryCurrentReply:
10
11        battery_current_json = pijuice.status.GetBatteryCurrent()
12
13        return deviceproperties_pb2.BatteryCurrentReply(
14            current = int(battery_current)
15        )
```

5. Back on the *monitoring-server* side, the battery_current_response variable, now updated with the most current metric, has its value put into the PostgreSQL database (Listing 7).

Listing 7 – Insertion SQL command and custom execute to enter data into the Postgresql database.

```
1 sql = [
2 "INSERT INTO node_parameters(node_uuid, battery_current) VALUES
      (%s, %s);"]
3 data = (
4         uuid.UUID(uuid_response.uuid),
5         battery_current_response.current
6         )
7 db.execute_query(sql, data)
```

Figure 12 – Flowchart illustrating how the information is gathered between the server and client.



Source: by the Author.

3.7 PACKAGING OF APPLICATIONS INTO CONTAINERS

The last step in the development of the monitoring applications is to containerize the server, client and database, making them suitable for a Kubernetes deployment; that is, the monitoring applications will be a deployment on the Kubernetes cluster itself, in the same manner as manner as the "other deployments" in Figure 9;

To this end, the Dockerfile reproduced in Listing 8 was created, using the Alpine Linux operating system as a base, chosen for its lean footprint and very good selection of packages for the *arm32v7* architecture, which is the target architecture for most of the Nodes residing in the Edge Layer. An auxiliary file was also used to create the container, which lists all the Python dependencies for the monitoring applications to run, defined in Listing 17.

Listing 8 – Dockerfile for the monitoring applications based on the Alpine Linux Distribution.

```
1  FROM alpine:3.16
2
3  ENV PYTHONUNBUFFERED=1
4  RUN apk add --update --no-cache python3 python3-dev make gcc
      musl-dev linux-headers && ln -sf python3 /usr/bin/python
5  RUN python3 -m ensurepip
6  RUN pip3 install --no-cache --upgrade pip setuptools make
7
8  WORKDIR /app
9  COPY . .
10
11 RUN pip install -r requirements.txt
12
13 CMD [ "python3", "monitoring_agent.py"]
```

For effects of comparison between the Kubernetes and the docker-compose approaches, a Docker Compose file was written that enables the two services on the same host machine. The *container:postgres* is the database container already running on the background. Particular notice should be taken of the volume mount, which mounts the `/etc/node_uuid` file from the host machine (which when deployed will be a Node running Linux on the Edge Layer) to the same file inside the container. This works as our Node identification mechanism.

Listing 9 – Cap.

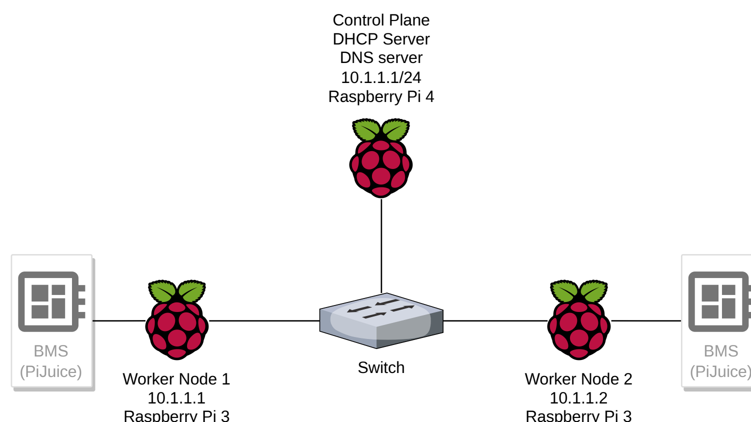
```
1 version: "3"
2 services:
3   client:
4     container_name: client
5     network_mode: "container:postgres"
6     image: monitoring-server
```

```
7       depends_on:
8       - "server"
9     server:
10      volumes:
11        - /etc/node_uuid:/etc/node_uuid
12      container_name: server
13      network_mode: "container:postgres"
14      image: monitoring_server
```

The Dockerfile for the *monitoring-server* is the same, but executing the corresponding Python file instead.

3.8 PHYSICAL CLUSTER

Figure 13 – Topology of the physical system.



Source: by the Author.

3.8.1 Hardware and Networking

For the hardware setup, a series of Raspberry Pi Single Board Computers (SBC), two Raspberry Pis Model 3 B+, and one Raspberry Pi Model 3 Model B were chosen. The Raspberry Pis Model 3 B+ were equipped with the PiJuice HAT™, which is a Raspberry Pi that can power the SBC using a battery with an integrated Battery Management System (BMS). The BMS provides information about the Node's power status (such as battery capacity, current draw...) using the I2C protocol, which can be read using the Pi[1].

To connect the Raspberry Pis, a NETGEAR ProSafe 5 Port Gigabit Switch, model GS105v5 was used. The topology is quite simple and the Raspberry Pis are connected locally via this switch using the Ethernet interface and are connected to the internet using the wireless

[1] Special thanks to my advisor for kindly letting one of the cabinets in his office be used as a test-bench for the hardware.

interface using the internationally available academic Wi-Fi network, Eduroam.

Thus, the Raspberry Pi 4 in this context represents an Edge Cloud Node and the Raspberry Pis 3 represent battery powered Edge Layer Node from the architecture described in 3.1.

The Raspberry Pi 4 runs the ISC DHCP server configured as in Listing 3.8.1, giving the Raspberry Pis 3 a fixed address each using a MAC Address filter.

```
1   default-lease-time 600;
2   max-lease-time 7200;
3   authoritative;
4
5   subnet 10.0.0.0 netmask 255.255.255.0 {
6       range 10.0.0.0 10.0.0.100;
7   }
8   host rpi1 {
9       hardware ethernet c7:35:ce:fd:8e:a1;
10      fixed-address 10.0.0.1;
11  }
12
13  host rpi2 {
14      hardware ethernet c7:35:ce:fd:8e:a1;
15      fixed-address 10.0.0.2;
16  }
```

3.9 SOFTWARE SETUP

For the software setup, it was decided to use one of the so-called "Kubernetes Distributions", in particular, K3s, due to its small size, having both ARM64 and ARMv7 as first-class citizens and simplified binary installation, containing all the necessary components for a complete Kubernetes-compatible orchestration system within less than 60MB.

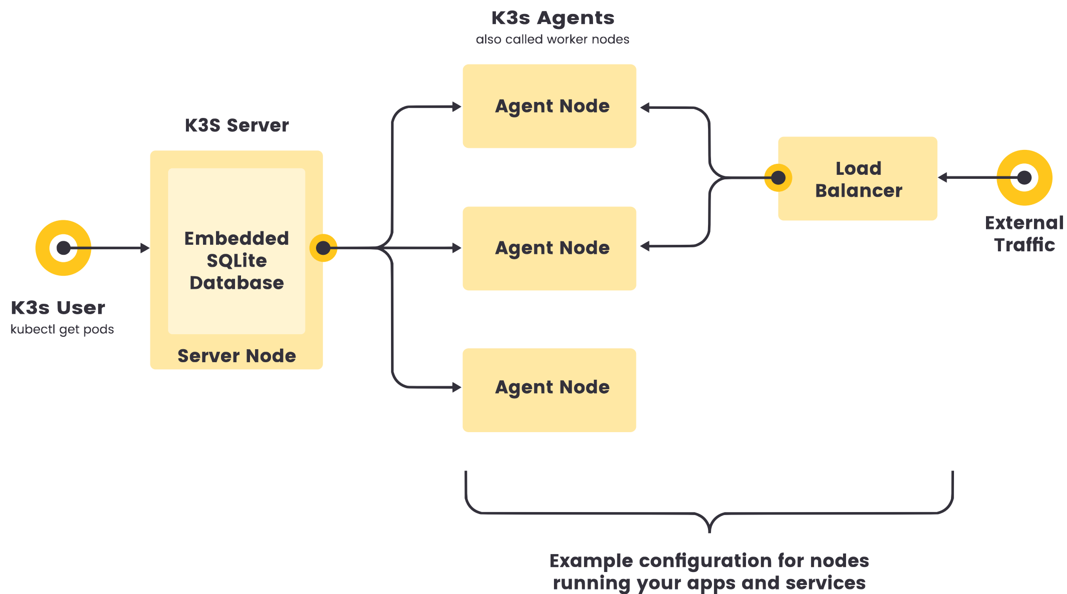
With the network configuration done, we ran the necessary commands to setup a single-server with an embedded database cluster as instructed in the K3s documentation (K3S, 2022). Figure 14 shows the architecture from the K3s point-of-view, with the "K3s server" and "K3s Agents" also being run on the physical "Control Plane" and Worker Nodes from figure 13 .

3.9.1 Performance considerations and benchmarking

Experimental data composed of the memory and CPU usage of the system while starting three replicas of a *nginx* server was gathered from the system using *Perl* and *bash* scripts, which are described in Appendix B.

The results corroborate with previous profiling studies such as (BÖHM; WIRTZ, 2021). It is possible to see that the memory (Figure 15) and CPU (Figure 16) usage have three spikes when spinning the service up but tend to stabilize over the course of a few seconds.

Figure 14 – Single-server Setup with an Embedded Database.



Source: K3s official documentation (K3S, 2022).

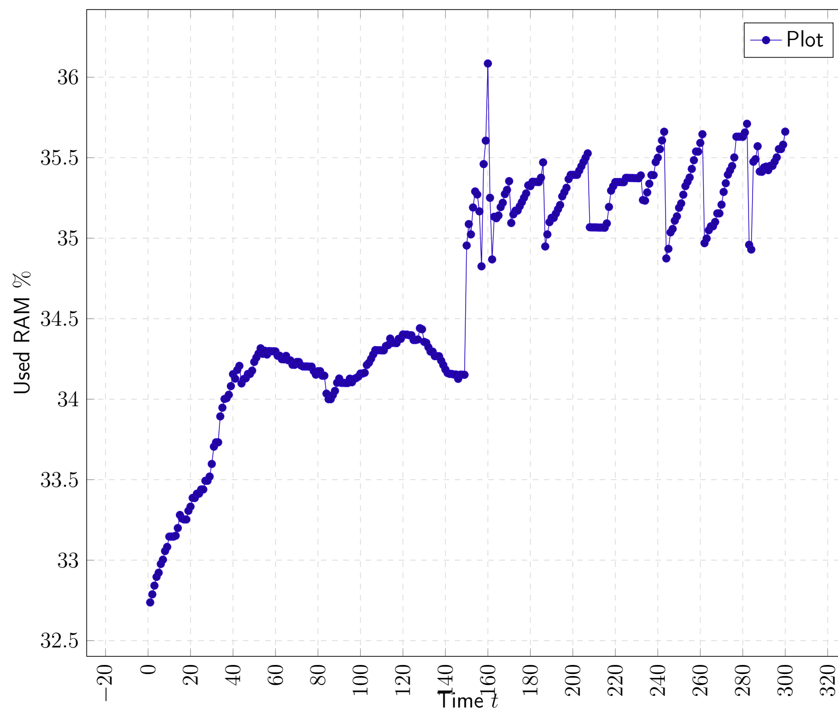


Figure 15 – Volatile Memory consumption over time.

Comparing the results with a native deployment of the *nginx* within a Unix operating system shows one of the downsides with this approach, as the RAM overhead is much greater than just the service running inside a Docker container without the orchestrator overhead, as it can be seen in Listing 10.

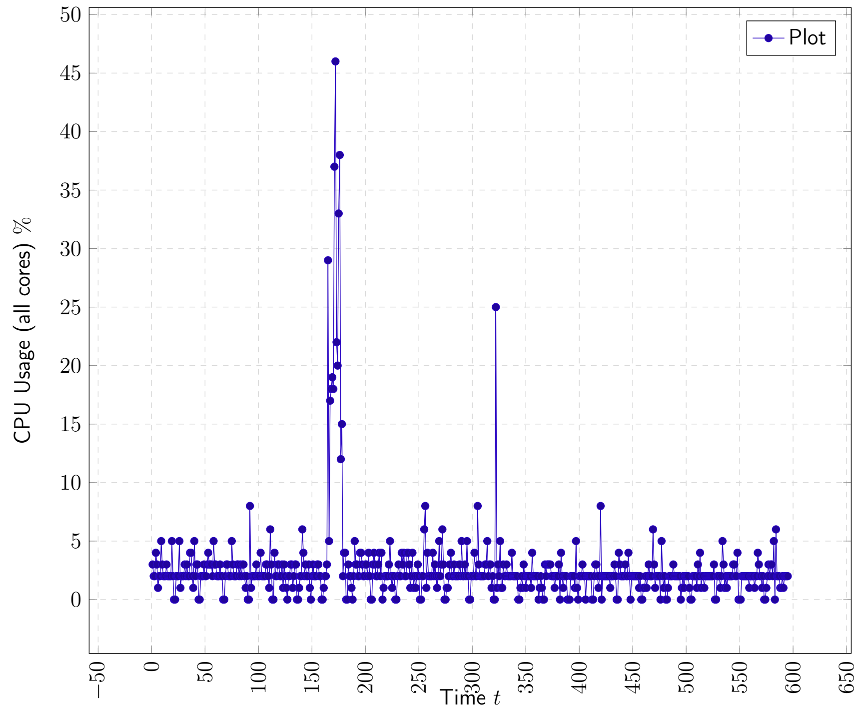


Figure 16 – Volatile Memory consumption over time.

Listing 10 – Memory and CPU usage of a nginx service running inside a Docker container, not considering the orchestrator overhead.

```
1
2       CPU %                   MEM USAGE
3       0.05%                    7.504MiB
```

3.10 CHANGING THE CLUSTER STATE BASED ON CONTEXT INFORMATION

With the cluster completely up and running on physical hardware, it is a matter of writing the proper YAML files to deploy the monitoring server and agent applications themselves on the cluster. A complete description of the code can be found in appendix. Having the monitoring applications running on the cluster enables a path of information between the Raspberry Pi Nodes and the database running on the server.

Getting information from the individual Nodes in the cluster is a first step providing a control loop over the state of cluster, but our implementation is still lacking a way to connect a software entity that monitors the cluster using the information provided by the Monitoring framework via the database and also issues commands to the Kubernetes cluster when actions must be taken.

To achieve such functionality, Kubernetes provides a REST API that can be used to control and receive information about the current state of the cluster. Instead of writing a higher-level logic to handle the REST API, the Kubernetes project support several classes in different programming languages that implement such communication with the API. As the

Monitoring framework is fully written in Python, the Kubernetes Client API implementation in Python was chosen.

As the monitoring framework was developed in a highly object-oriented manner, connecting the Kubernetes Python Client API to the database using the same connector class developed for the use within the monitoring server itself. Implementation details can be seen in the Appendix

And with this we have a complete monitoring solution of the each Node that is being deployed in the cluster.

3.11 USE-CASES FOR THE MONITORING FRAMEWORK

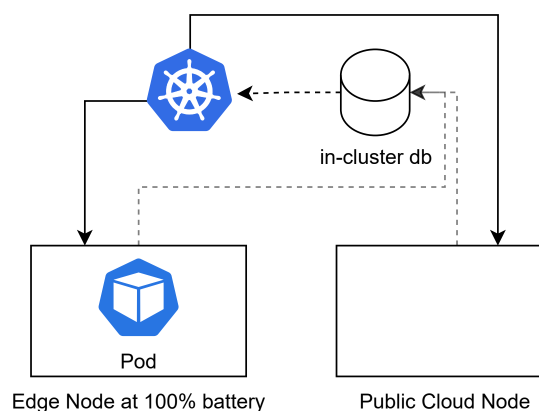
3.11.1 Use-case number 1 for the Monitoring framework: Context-Reactive Pod Switch

One interesting application of using Kubernetes in this context is that the combination Kubernetes and the Monitoring framework may be used to arrange custom switching between applications on the different layers. The use case scenario is the following:

1. Application runs on battery-powered Edge Layer Node.
2. Application consumes too much battery, in direction of a downtime.
3. Custom loop using reads the Monitoring framework database detects battery anomaly.
4. Custom loop issues a Kubernetes command to switch one or more Pods from the Edge Layer to the Public Cloud or Edge Cloud Layer.

A graphical explanation of the behaviour follows in figures 17 through 19.

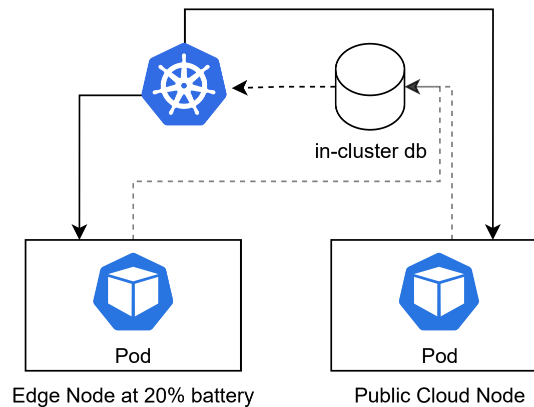
Figure 17 – Edge Node has a running Pod and has 100% battery capacity.



Source: by the Author.

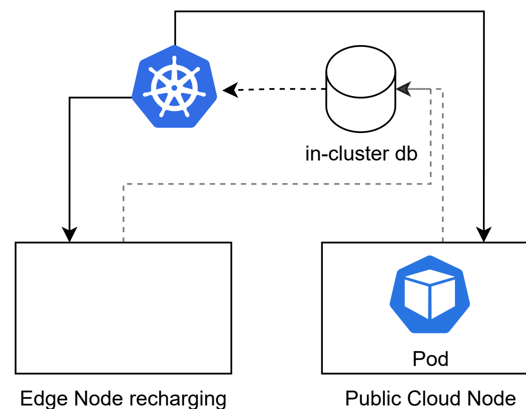
This is a direct application of a known pattern used in Kubernetes: automatic computational scaling and offloading (for example, running AI inference routines on Nodes with more

Figure 18 – Node falls to 20% battery, orchestrator automatically detects this through the monitoring framework and creates a Pod on the Public Cloud Layer.



Source: by the Author.

Figure 19 – A service watching the database, with battery information fed by the monitoring framework, switches the Pod to run in the Cloud, letting the Node recharge its battery or be serviced



Source: by the Author.

appropriate hardware), but with the augmented functionality of being context-aware, reacting to context and physical changes in the parameters of each Node.

The actual code that provides this functionality can be examined in Listing 18 of Appendix A, with the crucial snippet being a call to a `patch_deployment` function, which modifies the PodSpec to switch the application if a certain condition is met. In the original demonstrator, the case for switching a Pod was if the battery capacity went down by 30%, as it can be seen in the snippet. The battery capacity was taken directly from the database fed by the Monitoring framework.

Listing 11 – Example of a Kubernetes data description, in this case, a deployment of the nginx web server.

```
1 def patch_deployment(a, deployment):
2     api_response = a.patch_namespaced_deployment(
3             name = deployment,
4             namespace = 'default',
```

```
5              body = {"spec": {"template": {"spec": {"nodeSelector":
                  {"power": "low"}}}}},
6              pretty = 'true'
7          )
8      print("Deployment updated. status='%s'" %
          str(api_response.status))
9
10
11 if(last_battery_charge < 30):
12   patch_deployment()
```

3.11.2 Use-case number 2 for the Monitoring framework: Context-Aware Scheduling

This is a direct application from (CHIMA OGBUACHI et al., 2020). Using the approach on the aforementioned paper, we can defined a Node score function based on the metrics taken from the Monitoring's framework database.

The node score function is defined as:

$$\text{Node Score} = \prod_{i=1}^{|P|} (\frac{p_{min}}{p_i})^{-w_i} \tag{1}$$

Where:

- P is the number of parameters (in our case, system metrics such as temperature, battery percentage etc).
- p_{min} is the lowest historical value for that metric.
- p_i is the current value of the product loop.
- w_i is the current weight for that specific p metric, defined as

$$w_i = \frac{\sigma_i^2}{u_{i,n}} \tag{2}$$

with σ being the variance and u being the mean of that value.

As noted in (CHIMA OGBUACHI et al., 2020), because the complexity of calculating the variance and mean statistics grow linearly with the number of metrics in each parameter, the Welford Statistics as described in (MACLAREN, 1970) are used to calculate the mean, variance and standard deviation instead, here reproduced in equations (3), where k is the iterating variable.

$$u_1 = p_1, \qquad u_k = u_{k-1} + \frac{(p_k - M_{k-1})}{k} \tag{3a}$$

$$S_1 = 0, \qquad S_k = S_{k-1} + (p_k - u_{k-1}) \times (p_k - M_k) \tag{3b}$$

These are directly translated to Python code with the `RunningStatistics` class, which can be fully examined in Appendix A with relevant excerpt in Listing 12.

Listing 12 – Online methods to calculate the mean and variance of a given set.

```
1 self.new_mean = self.previous_mean + (x - self.previous_mean) /
    self.n
2 self.new_variance = self.old_variance + (x - self.previous_mean) *
    (x - self.new_mean)
```

As the development is fully object-oriented, it is a matter of instantiating the RunningStatistics class and using it together within the Kubernetes API to select the best Node based on the custom score. The functionality is encapsulated in the best_node method (Listing 13).

Listing 13 – Method to establish the best node giving a wide range of parameters.

```
1
2 def best_node(nodes):
3     if not nodes:
4         return []
5     node_scores = []
6     for node in nodes:
7         node_scores.append(get_node_score(node))
8     best_node = nodes[node_scores.index(min(node_scores)) + 1]
9     return best_node
```

A full example using the method described in this section, the Kubernetes API and the Database fed by the monitoring agents to schedule Pods to nodes can be found in Listing 20 within Appendix A.

4 CONCLUSION

4.1 DISCUSSION

The main objective stated in the introduction chapter was met, with the provided documentation and code this work provides a starting point for whoever wishes to created a distributed IoT project using Kubernetes and containerization technologies. The architecture is sufficiently adaptable such that any programming language, database or even orchestrator can be used and still benefit from the provided structure.

Regarding the specific objects, this work provided a brief historical overview of the containerization and orchestration ecosystem, technologies ubiquitous in the software engineering organizations today. We also have successfully implemented a monitoring framework and created some level of integration around it, including use-cases only present in academic works. The benchmarks also provide a safety backdrop for projects willing to use orchestration systems in production, which seems now reminiscent of the early days of the containerization approach adoption for embedded devices.

4.2 FUTURE WORKS

Future works may explore areas such as:

- Use of the remoteproc or RPMsg, Linux Kernel frameworks that can reprogram or communicate directly with microcontrollers, connecting the Edge Layer and Smart Sensor Layer directly, as an alternative deployment scenario.
- Variable period of sampling for the different system metrics.
- Use of off-the-shelf observability tools such as fluentbit instead of the custom monitoring framework.
- Continuous Delivery/Continuous Integration within the cluster for automatic IoT updates, which is arguably one of the most sought-after commercial areas in embedded Linux.
- Conduct studies on the resource consumption modeling of different orchestrators.

REFERENCES

ATZORI, Luigi; IERA, Antonio; MORABITO, Giacomo. The Internet of Things: A survey. **Computer Networks**, v. 54, n. 15, p. 2787–2805, 2010. ISSN 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2010.05.010. Available from: <https://www.sciencedirect.com/science/article/pii/S1389128610001568>. Cit. on pp. 7, 18.

BÖHM, Sebastian; WIRTZ, Guido. Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes. In: ZEUS. [S.l.: s.n.], 2021. P. 65–73. Cit. on p. 41.

BUGNION, Edouard; NIEH, Jason; TSAFRIR, Dan. **Hardware and Software Support for Virtualization**. [S.l.]: Springer International Publishing, 2017. DOI: 10.1007/978-3-031-01753-7. Available from: <https://doi.org/10.1007/978-3-031-01753-7>. Cit. on pp. 19–21.

BURNS, Brendan et al. Borg, omega, and kubernetes. **Communications of the ACM**, ACM New York, NY, USA, v. 59, n. 5, p. 50–57, 2016. Cit. on pp. 26, 27.

CASALICCHIO, Emiliano. Container Orchestration: A Survey. In: **Systems Modeling: Methodologies and Tools**. Ed. by Antonio Puliafito and Kishor S. Trivedi. Cham: Springer International Publishing, 2019. P. 221–235. ISBN 978-3-319-92378-9. DOI: 10.1007/978-3-319-92378-9_14. Available from: <https://doi.org/10.1007/978-3-319-92378-9_14>. Cit. on pp. 21, 22, 25.

CHIMA OGBUACHI, Michael et al. Context-aware Kubernetes scheduler for edge-native applications on 5G. **Journal of communications software and systems**, Udruga za komunikacijske i informacijske tehnologije, Fakultet . . ., v. 16, n. 1, p. 85–94, 2020. Cit. on pp. 7, 18, 28, 32, 46.

CNCF. **CNCF Annual Survey 2021**. [S.l.: s.n.], Aug. 2021. Cit. on p. 26.

GOOGLE. **Release 0.1.0 · Google/Lmctfy**. [S.l.]: Google, Oct. 2013. Available from: <https://github.com/google/lmctfy/releases/tag/0.1.0>. Cit. on p. 23.

HOCKIN, Tim. **lmctfy@googlegroups**. [S.l.]: Google, Oct. 2013. Available from: <https://groups.google.com/g/lmctfy/c/e6oGQELK2oA/m/1TEGXtRUdBAJ>. Cit. on p. 23.

JNAGAL, Rohit. **Update project status**. [S.l.]: Google, May 2015. Available from: <https://github.com/google/lmctfy/commit/0b317d7eb625d1877a8a0aaf2f46f770d9a5a50f>. Cit. on p. 23.

K3S. **K3s - Lightweight Kubernetes**. [S.l.: s.n.], 2022. Available from: <https://docs.k3s.io/>. Visited on: 15 July 2022. Cit. on pp. 41, 42.

LABORATORIES, Bell Telephone. **Unix Programmer's Manual**. 7. ed. [S.l.], Jan. 1979. Cit. on p. 22.

LAMMI, Toni Juhani. **Feasibility of application containers in embedded real-time Linux**. 2018. MA thesis. Cit. on p. 28.

LEE, Hyeongju et al. Scalable and High Available Kubernetes Cluster in Edge Environments for IoT Applications. Helsingin yliopisto, 2021. Cit. on p. 28.

LINUX. **cgroups(7) — Linux manual page**. [S.l.], Aug. 21. Cit. on p. 22.

LINUX. **namespaces(7) — Linux manual page**. [S.l.], Aug. 21. Cit. on pp. 22, 23.

LXC. **What's LXC?** [S.l.: s.n.]. LXC Website. Available from: <https://linuxcontainers.org/lxc/introduction/>. Cit. on p. 22.

MACLAREN, M Donald. The art of computer programming. Volume 2: Seminumerical algorithms (Donald E. Knuth). **SIAM Review**, SIAM, v. 12, n. 2, p. 306–308, 1970. Cit. on p. 46.

MORABITO, Roberto; KJÄLLMAN, Jimmy; KOMU, Miika. Hypervisors vs. Lightweight Virtualization: A Performance Comparison. In: 2015 IEEE International Conference on Cloud Engineering. [S.l.: s.n.], 2015. P. 386–393. DOI: 10.1109/IC2E.2015.74. Cit. on p. 21.

NORONHA, Vivian et al. Performance evaluation of container based virtualization on embedded microprocessors. In: IEEE. 2018 30th International Teletraffic Congress (ITC 30). [S.l.: s.n.], 2018. P. 79–84. Cit. on p. 28.

OCI. **Open Containers Overview**. [S.l.]: OCI, June 2015. Available from: <https://opencontainers.org/about/overview/>. Cit. on p. 24.

POPEK, Gerald J.; GOLDBERG, Robert P. Formal Requirements for Virtualizable Third Generation Architectures. **Commun. ACM**, Association for Computing Machinery, New

York, NY, USA, v. 17, n. 7, p. 412–421, July 1974. ISSN 0001-0782. DOI: 10.1145/361011.361073. Available from: <https://doi.org/10.1145/361011.361073>. Cit. on p. 19.

RAD, Babak Bashari; BHATTI, Harrison John; AHMADI, Mohammad. An introduction to docker and analysis of its performance. **International Journal of Computer Science and Network Security (IJCSNS)**, International Journal of Computer Science and Network Security, v. 17, n. 3, p. 228, 2017. Cit. on pp. 21, 23.

STATISTA. **Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030**. [S.l.: s.n.], 2022. Available from: <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/>. Visited on: 15 Jan. 2023. Cit. on p. 18.

Appendix

APPENDIX A – MONITORING FRAMEWORK CODE AND DEPLOYMENT FILES

A.1 FRAMEWORK CODE

Listing 14 – Code for the *monitoring-server* portion of the framework seen in Figure 9.

```
1  #!/usr/bin/python3
2  import asyncio
3  import logging
4
5  from database_setup import Database
6
7  import grpc
8  import uuid
9  from random import randrange
10
11 # these should be generated by our protobuf definition
12 import deviceproperties_pb2
13 import deviceproperties_pb2_grpc
14
15 async def run() -> None:
16
17   while True:
18     async with grpc.aio.insecure_channel("monitoring-agent:50051")
         as channel:
19       stub =
            deviceproperties_pb2_grpc.GetDeviceInformationStub(channel)
20
21       try:
22         battery_percentage_response =
23           await stub.BatteryPercentage(
24             deviceproperties_pb2
25             .google_dot_protobuf_dot_empty__pb2.Empty())
26
27         battery_voltage_response =
28           await stub.BatteryVoltage(
29             deviceproperties_pb2
30             .google_dot_protobuf_dot_empty__pb2.Empty())
31
32         battery_current_response =
33           await stub.BatteryCurrent(
```

```
34              deviceproperties_pb2
35              .google_dot_protobuf_dot_empty__pb2.Empty())
36
37          gpio_voltage_response =
38            await stub.GpioVoltage(
39              deviceproperties_pb2
40              .google_dot_protobuf_dot_empty__pb2.Empty())
41
42          gpio_current_response =
43            await stub.GpioCurrent(
44              deviceproperties_pb2
45              .google_dot_protobuf_dot_empty__pb2.Empty())
46
47          uuid_response =
48            await stub.Uuid(
49              deviceproperties_pb2.google_dot_protobuf_dot_empty__pb2.Empty())
50
51          logging.info("GetDeviceInformation client received:")
52          logging.info("BatteryPercentage: " +
               str(battery_percentage_response.percentage))
53          logging.info("BatteryVoltage: " +
               str(battery_voltage_response.voltage))
54          logging.info("BatteryCurrent: " +
               str(battery_current_response.current))
55          logging.info("GpioVoltage: " +
               str(gpio_voltage_response.voltage))
56          logging.info("GpioCurrent: " +
               str(gpio_current_response.current))
57          logging.info("Device UUID: " + uuid_response.uuid)
58
59        except grpc.RpcError as e:
60          logging.info("Agents are probably not up, just retry...")
61          pass
62
63      logging.info("Inserting actual values from the battery")
64      sql = [
65        "INSERT INTO node_parameters(node_uuid, battery_percentage,
             battery_voltage, battery_current, gpio_voltage,
             gpio_current) VALUES (%s, %s, %s, %s, %s, %s);"]
66      data = (
67          uuid.UUID(uuid_response.uuid),
68          battery_percentage_response.percentage,
```

```
69          battery_voltage_response.voltage,
70          battery_current_response.current,
71          gpio_voltage_response.voltage,
72          gpio_current_response.current)
73      db.execute_query(sql, data)
74  
75      await asyncio.sleep(1)
76  
77  if __name__ == "__main__":
78    logging.basicConfig(level=logging.INFO)
79  
80    db = Database(host="postgres", user="postgres", dbname=None)
81    db.connect()
82    logging.info("Connected to database")
83    db.create_database(dbname="database")
84  
85    logging.info("Creating node_parameters table")
86    db.create_tables(
87        [
88            """
89              CREATE TABLE node_parameters (
90              id SERIAL PRIMARY KEY,
91              node_uuid UUID NOT NULL,
92              battery_percentage VARCHAR(255) NOT NULL,
93              battery_voltage VARCHAR(255) NOT NULL,
94              battery_current VARCHAR(255) NOT NULL,
95              gpio_voltage VARCHAR(255) NOT NULL,
96              gpio_current VARCHAR(255) NOT NULL,
97              inserted_at TIMESTAMP WITH TIME ZONE DEFAULT
                  CURRENT_TIMESTAMP
98              )
99          """
100       ]
101   )
102 
103   logging.basicConfig()
104   loop = asyncio.get_event_loop()
105   while True:
106     loop.run_until_complete(run())
```

Listing 15 – Code for the *monitoring-agent* portion of the framework seen in Figure 9.

```
1 #!/usr/env/python3
```

```
2  from pijuice import PiJuice
3  import json
4
5  import asyncio
6  import logging
7
8  import grpc
9
10 import deviceproperties_pb2
11 import deviceproperties_pb2_grpc
12
13 pijuice = PiJuice(1, 0x14)
14
15 class GetDeviceInformation(
16   deviceproperties_pb2_grpc.GetDeviceInformationServicer):
17   async def BatteryPercentage(
18       self,
19       request:
           deviceproperties_pb2.google_dot_protobuf_dot_empty__pb2.Empty(),
20       context: grpc.aio.ServicerContext,
21   ) -> deviceproperties_pb2.BatteryPercentageReply:
22     charge_percentage = pijuice.status.GetChargeLevel()
23
24     return deviceproperties_pb2.BatteryPercentageReply(
25         percentage=int(charge_percentage[data])
26     )
27
28   async def BatteryVoltage(
29     self,
30     request:
          deviceproperties_pb2.google_dot_protobuf_dot_empty__pb2.Empty(),
31     context: grpc.aio.ServicerContext,
32   ) -> deviceproperties_pb2.BatteryVoltageReply:
33
34     battery_voltage_json = pijuice.status.GetBatteryVoltage()
35
36     return deviceproperties_pb2.BatteryVoltageReply(
37         voltage=int(battery_voltage_json.[data])
38     )
39
40   async def BatteryCurrent(
41     self,
```

```
42    request:
        deviceproperties_pb2.google_dot_protobuf_dot_empty__pb2.Empty(),
43    context: grpc.aio.ServicerContext,
44  ) -> deviceproperties_pb2.BatteryCurrentReply:
45
46    battery_current_json = pijuice.status.GetBatteryCurrent()
47
48    return deviceproperties_pb2.BatteryCurrentReply(
49        current=int(battery_current_json.[data])
50    )
51
52  async def GpioVoltage(
53    self,
54    request:
        deviceproperties_pb2.google_dot_protobuf_dot_empty__pb2.Empty(),
55    context: grpc.aio.ServicerContext,
56  ) -> deviceproperties_pb2.GpioVoltageReply:
57
58    gpio_voltage_json = pijuice.status.GetIoVoltage()
59
60    return deviceproperties_pb2.GpioVoltageReply(
61        voltage=int(gpio_voltage_json.[data])
62    )
63
64  async def GpioCurrent(
65    self,
66    request:
        deviceproperties_pb2.google_dot_protobuf_dot_empty__pb2.Empty(),
67    context: grpc.aio.ServicerContext,
68  ) -> deviceproperties_pb2.GpioCurrentReply:
69
70    gpio_current_json = pijuice.status.GetIoCurrent()
71
72    return deviceproperties_pb2.GpioCurrentReply(
73        current=int(gpio_current_json[data])
74    )
75
76  async def Uuid(
77    self,
78    request:
        deviceproperties_pb2.google_dot_protobuf_dot_empty__pb2.Empty(),
79    context: grpc.aio.ServicerContext,
```

```
80   ) -> deviceproperties_pb2.UuidReply:
81
82     f = open("/etc/node_uuid", "r")
83     node_uuid = str(f.readline()).rstrip()
84     return deviceproperties_pb2.UuidReply(
85         uuid=str(node_uuid)
86     )
87
88
89 async def serve() -> None:
90   server = grpc.aio.server()
91   deviceproperties_pb2_grpc.add_GetDeviceInformationServicer_to_server(
92       GetDeviceInformation(), server
93   )
94   listen_addr = "[::]:50051"
95   server.add_insecure_port(listen_addr)
96   logging.info("Starting server on %s", listen_addr)
97   await server.start()
98   await server.wait_for_termination()
99
100
101 if __name__ == "__main__":
102   logging.basicConfig(level=logging.INFO)
103   asyncio.run(serve())
```

Listing 16 – Database class used to create a database, connect, execute commands, queries and table creation and destruction on a PostgreSQL database instance.

```
1  #!/usr/env/python
2
3  import psycopg2
4  import psycopg2.extras
5
6
7  class Database:
8    """
9    This database connector DOES NOT SANITIZES THE INPUTS.
10   PLEASE DON'T DEPLOY THIS ANYWHERE NEAR AN END USER.
11   IT IS FOR PURELY FOR PROOF OF CONCEPTS THAT NEED A DB.
12   DO NOT DEPLOY IN ANY WAY, SHAPE OR FORM.
13   """
14
15   def __init__(self, host="localhost", user="postgres",
```

```
         dbname="database"):
16     # FIXME(ljh): these values should be initilized by connect()
17     self.conn = None
18     self.cursor = None
19     self.host = host
20     self.user = user
21     self.dbname = dbname
22
23   def connect(self, dbname=None):
24     psycopg2.extras.register_uuid()
25     try:
26       self.conn = psycopg2.connect(
27           host=self.host, user=self.user, dbname=dbname,
              password="postgres"
28       )
29       if self.conn is None:
30         print(
31             "Connection could not be established, connection handler
                is "
32             + self.conn
33         )
34       self.conn.set_session(autocommit=True)
35       self.cursor = self.conn.cursor()
36       print("Version:")
37       version = self.cursor.execute("SELECT version()")
38       row = self.cursor.fetchone()
39       if row is None:
40         raise ValueError(
41             "Cursor could not execute the request of SELECT
                version()"
42         )
43       print(row)
44
45     except (Exception, psycopg2.DatabaseError) as error:
46       print(error)
47     except ValueError as err:
48       print(err.args)
49
50   def destroy_connection(self):
51     self.conn = None
52
53   def create_tables(self, commands):
```

```
54        try:
55          for command in commands:
56            self.cursor.execute(command)
57          self.conn.commit()
58        except (Exception, psycopg2.DatabaseError) as error:
59          print(error)
60        finally:
61          if self.conn is not None:
62            self.conn.close()
63
64      def execute_query(self, queries, data=None):
65        """
66        I know, double try-catch hell. But this is the only way to make
             sure that
67        if a connection gets dropped, it will try back again.
68        queries is a list of queries, like this:
69        ["query1", "query2", "query3", "query4"]
70        Make sure that if you pass a single query, it is an 1-ary list.
71        """
72        for query in queries:
73          try:
74            if (data is None):
75              print("Data is None, please use %s if that's an insert")
76              self.cursor.execute(query)
77            else:
78              self.cursor.execute(query, data)
79            self.conn.commit()
80          except (Exception, psycopg2.DatabaseError) as error:
81            print(error)
82            try:
83              self.cursor.close()
84              self.cursor = self.conn.cursor()
85            except BaseException:
86              self.conn.close()
87              psycopg2.extras.register_uuid()
88              self.conn = psycopg2.connect(
89                  host=self.host,
90                  user=self.user,
91                  dbname=self.dbname,
92                  password="postgres")
93              if self.conn is None:
94                print(
```

```
95                "Connection could not be established, connection
                     handler is "
96                + self.conn
97             )
98         self.cursor = self.conn.cursor()
99         if (data is None):
100          self.cursor.execute(query)
101        else:
102          self.cursor.execute(query, data)
103        self.conn.commit()
104
105   def create_database(self, dbname="database"):
106     # checks if database "database" already exists
107     self.cursor.execute(
108       "SELECT 1 FROM pg_catalog.pg_database WHERE datname =
             'database'")
109     exists = self.cursor.fetchone()
110     if not exists:
111       self.execute_query(['CREATE DATABASE database'])
112       self.conn.commit()
```

Listing 17 – requirements.txt file, which contains the dependencies for the monitoring python applications.

```
1 grpcio==1.49.0
2 protobuf==4.21.5
3 psycopg2-binary==2.9.3
4 six==1.16.0
5 smbus==1.1.post2
```

Listing 18 – Example of context-reactive Pod switch using the database and Kubernetes APIs.

```
1 #!/usr/bin/env python
2 from database_setup import Database
3 from kubernetes import client, config
4
5 # Configs can be set in Configuration class directly or using helper
     utility
6 config.load_kube_config()
7
8 kubernetes_api = client.AppsV1Api()
9
10 db = Database(host="localhost", user="postgres", dbname=None)
11 db.connect()
```

```
12 print("Connected to database")
13
14 # grabs the last row on node_parameters
15 db.cursor.execute(
16     "SELECT * FROM node_parameters WHERE id=(SELECT max(id) FROM
        node_parameters);")
17 row = db.cursor.fetchone()
18 last_battery_charge = row[2]
19 print("Last battery percentage is: " + last_battery_charge)
20
21 def patch_deployment(a, deployment):
22     api_response = a.patch_namespaced_deployment(
23             name = deployment,
24             namespace = 'default',
25             body = {"spec": {"template": {"spec": {"nodeSelector":
                   {"power": "low"}}}}},
26             pretty = 'true'
27         )
28     print("Deployment updated. status='%s'" %
        str(api_response.status))
29
30
31 if(last_battery_charge < 30):
32   patch_deployment()
33 else:
34   pass
```

Listing 19 – RunningStatistics class.

```
 1 from cmath import sqrt
 2
 3
 4 class RunningStatistics:
 5   """
 6   Usage:
 7   ```
 8     rv = RunningStatistics()
 9     rv.insert(17.0)
10     rv.insert(19.0)
11     rv.insert(24.0)
12     mean = rv.mean();
13     variance = rv.variance();
14     stdev = rv.standard_deviation();
```

```
15  ```
16  every time `insert` is called, the values are automatically
       computed with an online method.
17  You can test the accuracy of the online method comparing it with
       numpy's own:
18  ```
19    print("Using Numpy Methods:")
20    arr = [17.0, 19.0, 24.0]
21    import numpy as np
22    print("Mean: ", np.mean(arr))
23    # see https://www.embeddedrelated.com/showarticle/785.php
24    print("Variance: ", np.var(arr, ddof=1))
25    print("Standard Deviation: ", np.std(arr, ddof=1))
26  ```
27  """
28  n = 0
29  previous_mean = 0.0
30  new_mean = 0.0
31  old_variance = 0.0
32  new_variance = 0.0
33
34  def insert(self, x):
35    self.n += 1
36
37    if (self.n == 1):
38      self.previous_mean = self.new_mean = x
39      self.old_variance = 0.0
40    else:
41      self.new_mean = self.previous_mean + (x - self.previous_mean)
          / self.n
42      self.new_variance = self.old_variance + \
43          (x - self.previous_mean) * (x - self.new_mean)
44
45      self.previous_mean = self.new_mean
46      self.old_variance = self.new_variance
47
48  def mean(self):
49    if self.n > 0:
50      return self.new_mean
51    else:
52      return 0.0
53
```

```
54   def variance(self):
55     if self.n > 1:
56       return self.new_variance / (self.n - 1)
57     else:
58       return 0.0
59
60   def standard_deviation(self):
61     return sqrt(self.variance())
```

Listing 20 – Complete example using a custom node ranking algorithm using information from the database of the monitoring application and the Kubernetes API.

```
1  #!/usr/bin/env python
2  from database_setup import Database
3  from kubernetes import client, config, watch
4  from welford import RunningStatistics
5
6  import json
7
8  config.load_kube_config()
9
10 kubernetes_api = client.AppsV1Api()
11
12 db = Database(host="localhost", user="postgres", dbname=None)
13 db.connect()
14 print("Connected to database")
15
16 def get_parameter_statistics(data):
17   rs = RunningStatistics()
18
19   #FIXME(ljh): AFTER this is done once, then it should only get the
       LAST result from the database, because it's an online method.
       Getting everything like this right now is EXTREMELY DUMB.
20   for x in data:
21     rs.insert(x)
22
23   mean = rs.mean()
24   variance = rs.variance()
25   weight = variance / mean
26   return weight
27
28 def get_node_score(node):
29   # grabs the last row on node_parameters
```

```
30   query = "SELECT * FROM node_parameters WHERE hostname =
       '{}';".format(node.metadata.name)
31   print("Querying information from node")
32   print("Query:", query)
33   db.cursor.execute(query)
34   rows = db.cursor.fetchall()
35
36   battery_percentage = []
37   for row in rows:
38     battery_percentage.append(row[2])
39     """
40     current.append(row[...
41     """
42   weight = get_parameter_statistics(battery_percentage)
43
44   # This is not considering the summation, only one parameter
45   node_score = (min(battery_percentage) /
46                 battery_percentage[-1]) ** (- weight)
47   return node_score
48
49 def best_node(nodes):
50   if not nodes:
51     return []
52   node_scores = []
53   for node in nodes:
54     node_scores.append(get_node_score(node))
55   best_node = nodes[node_scores.index(min(node_scores)) + 1]
56   return best_node
57
58 def nodes_available():
59   ready_nodes = []
60   for n in kubernetes_api.list_node().items:
61     # This loops over the nodes available. n is the node. We are
         trying to
62     # schedule the pod on one of those nodes.
63     for status in n.status.conditions:
64       if status.status == "True" and status.type == "Ready":
65         ready_nodes.append(n.metadata.name)
66   return ready_nodes
67
68 def scheduler(name, node, namespace="default"):
69   target = client.V1ObjectReference()
```

```
70    target.kind = "Node"
71    target.apiVersion = "v1"
72    target.name = node
73    meta = client.V1ObjectMeta()
74    meta.name = name
75    body = client.V1Binding(target=target)
76    body.metadata = meta
77    return kubernetes_api.create_namespaced_binding(
78        namespace, body, _preload_content=False)
79
80  def main():
81    # Watch for events
82    w = watch.Watch()
83    for event in w.stream(kubernetes_api.list_namespaced_deployment,
         "default"):
84      # and event['object'].spec.scheduler_name == scheduler_name:
85      if event['object'].status.phase == "Pending":
86        try:
87          chosen_node = best_node(nodes_available())
88          print("Scheduling " + event['object'].metadata.name)
89          res = scheduler(event['object'].metadata.name, chosen_node)
90        except client.rest.ApiException as e:
91          print("exception")
92          print(json.loads(e.body)['message'])
93
94  if __name__ == '__main__':
95    main()
```

A.2 DEPLOYMENT OF THE MONITORING FRAMEWORK ON KUBERNETES

As described in section 3.2, Kubernetes has its desired state represented by YAML files. We can use this to deploy the services that compose the Monitoring framework on the assembled cluster.

In Listing 21, we use the *DaemonSet* configuration, which makes sure every node in the cluster gets a monitoring-agent deployed on it.

Listing 21 – YAML file Kubernetes deployment description of the *monitoring-agent* application using *Daemonset* to deploy an agent on every node in the cluster.

```
1
2  apiVersion: apps/v1
3  kind: DaemonSet
4  metadata:
```

```
5    name: monitoring-agent
6    labels:
7      app: monitoring-agent
8  spec:
9    selector:
10     matchLabels:
11       name: monitoring-agent
12   template:
13     metadata:
14       labels:
15         name: monitoring-agent
16     spec:
17       imagePullSecrets:
18       - name: energy-aware-orchestrator-registry-key
19       containers:
20       - name: monitoring-agent
21         image: registry.gitlab.cc-asp.fraunhofer.de/leo35013/\
22         energy-aware-orchestrator/monitoring-agent
```

In a similar fashion the deploy of the *monitoring-server* can be done. As is shown in Listing 22, instead of deploying to every node in the cluster, we use the *matchLabels* property, which allows only nodes marked as "monitoring-server" to get this deployment assigned to them.

Listing 22 – YAML file Kubernetes deployment description of the *monitoring-server* application.

```
1
2  apiVersion: apps/v1
3  kind: Deployment
4  metadata:
5    name: monitoring-server
6  spec:
7    replicas: 1
8    selector:
9      matchLabels:
10       app: monitoring-server
11   template:
12     metadata:
13       labels:
14         app: monitoring-server
15     spec:
16       imagePullSecrets:
17       - name: energy-aware-orchestrator-registry-key
```

```
18        containers:
19          - image: registry.gitlab.cc-asp.fraunhofer.de/leo35013/\
20          energy-aware-orchestrator/monitoring-server
21            name: monitoring-server
22            env:
23            - name: POSTGRES_HOST
24              value: postgres
25            - name: POSTGRES_PORT
26              value: "5432"
27            - name: POSTGRES_DB
28              value: database
29            - name: POSTGRES_USER
30              value: postgres
31            - name: POSTGRES_PASSWORD
32              value: postgres
```

A.3 SIMULATING THE CLUSTER WITH *KIND*

The implementation of the monitoring described in section 3.2 was done regardless of any Orchestrator software in place. To test the functionality of the software without any physical hardware, we have used the tool *kind*, which runs a local Kubernetes cluster within a development computer using Docker Containers as "Virtual Nodes".

The configuration file in Listing 23 for the *kind* software was created. As with the real Kubernetes setup, the nodes representing the Edge Layer were given the "monitored" label, which signifies that an agent of the Monitoring framework has to run on that particular node. This was later done with Kubernetes deployment files, explained in the later sections of this chapter.

Listing 23 – YAML file describing a virtual cluster using *kind* with two worker nodes and one control plane node.

```
1   kind: Cluster
2   apiVersion: kind.x-k8s.io/v1alpha4
3   name: mock-cluster
4   nodes:
5   - role: control-plane
6   - role: worker
7   labels:
8   monitored: yes
9   - role: worker
```

As a helper to the cluster configuration in Listing 23, the bash script in Listing 24 was created. The result is a full virtual cluster that we can use to write our deployment files for

the monitoring solution we have developed. One important step that must be highlighted is the creation of a Kubernetes Secret (that can be seen being used on the deployment files in Listing 21 and Listing 22 in the *imagePullSecrets* spec), which makes Kubernetes able to pull images from the a private Docker Registry, which in our particular case was a managed registry hosted on GitLab.

Listing 24 – Script to start the *kind* virtual cluster.

```
kind create cluster --name mock --config cluster-config.yaml
kubectl cluster-info --context kind-mock
kubectl label node mock-worker node-role.kubernetes.io/worker=worker
kubectl label node mock-worker2
    node-role.kubernetes.io/worker=worker1

# every deployment needs to have the following: `imagePullSecrets:
    -name: energy-aware-orchestrator-registry-key` on the `spec:
    containers:` field.
kubectl create secret docker-registry
    energy-aware-orchestrator-registry-key \
  --docker-server=$DOCKER_REGISTRY_SERVER \
  --docker-username=$DOCKER_USER \
  --docker-password=$DOCKER_PASSWORD \
  --docker-email=$DOCKER_EMAIL
```

APPENDIX B – SCRIPTS USED TO GATHER RUNTIME PARAMETERS

Listing 25 – Perl script used to get data used to generate Figure 15.

```
1
2 # usage: bash memory-usage.sh <name of the file to be written>.txt
3 timeout="300"
4 counter="0"
5 cat /proc/meminfo | awk -vORS=, '{ print $1 } ' | sed 's/,$/\n/' >>
     $1
6 while [ ${counter} -ne ${timeout} ]
7 do
8 cat /proc/meminfo | awk -vORS=, '{ print $2 } ' | sed 's/,$/\n/' >>
     $1
9 ((counter++))
10 sleep 2
11 done
```

Listing 26 – Perl script used to get data used to generate figure Figure 16.

```
1
2     #!/usr/bin/perl
3
4     open VMSTAT, "vmstat 2|";
5     <VMSTAT>; <VMSTAT>; # skip the header
6     my $filename = '/home/pi/cpu-usage.csv';
7     while (<VMSTAT>) {
8         open(my $fh, '>>', $filename);
9         @now = split;
10        ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
              localtime (time);
11        printf $fh "%d\n", $now[12] + $now[13];
12    }
```

APPENDIX C – A PRIMER ON USING DOCKER AND DOCKER COMPOSE.

C.1 DOCKER

Images can be packaged and uploaded to a common web server called a Registry. As an example, we can use the Docker CLI (Command Line Interface) to pull an image from the default DockerHub registry, the result is printed in Listing 27.

Listing 27 – Pulling a ubuntu image from the default DockerHub registry using the Docker CLI.

```
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
301a8b74f71f: Pull complete
Digest: sha256:7cfe75438fc77c9d7235ae502 \
  bf229b15ca86647ac01c844b272b56326d56184
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

This image is now locally available on our system. If we analyze this image and list its entries[1] in Listing 28, it is possible to see that it is actually composed of metadata and a root filesystem.

Listing 28 – Files that compose a container image.

```
|-- 58db3edaf2be6e80f628796355b1bdeaf\
      8bea1692b402f48b7e7b8d1ff100b02.json
|-- d5e9028c535cebbd5d2243c08c5\
      1ba634d385964dc6b63987bb37c5b844c4140
|   |-- bin -> usr/bin
|   |-- boot
|   |-- dev
|   |-- etc
|   |-- home
|   |-- json
|   |-- lib -> usr/lib
|   |-- lib32 -> usr/lib32
|   |-- lib64 -> usr/lib64
|   |-- libx32 -> usr/libx32
|   |-- media
|   |-- mnt
|   |-- opt
|   |-- proc
```

[1] The image was saved using the docker save command as a tar file and then expanded.

```
19 |   |-- root
20 |   |-- run
21 |   |-- sbin -> usr/sbin
22 |   |-- srv
23 |   |-- sys
24 |   |-- tmp
25 |   |-- usr
26 |   |-- var
27 |   `-- VERSION
28 |-- manifest.json
29 `-- repositories
```

The file `58db3edaf2be6e80f628796355b1bdeaf8bea1692b402f48b7e7b8d1ff100b02.json` contains the necessary metadata that will used to create the starting configuration of a container from this image. For example, if we open the file it is possible to see that when the container is created from this image, the running process will be bash, due to the `"Cmd"` field on the json file (Listing 29).

Listing 29 – Excerpt from the metadata file highlighting the Cmd section.

```
1 "Cmd": [
2     "/bin/bash"
3 ],
```

And actually creating the container with the `"-interactive -tty"` options to keep `"STDIN"` alive and allocate a pseudo-TTY will confirm that the bash shell will actually be running process.

Listing 30 – A primer on using Docker and Docker Compose.

```
1     $  docker run --interactive --tty ubuntu
2     root@eb93e932034f:# echo $SHELL
3     /bin/bash
```

In Listing 31 it is also possible to see which namespaces (see discussion in section 2.3) were created in order to run the container.

Listing 31 – Namespaces created when running *bash* inside a Debian container

```
1 NS TYPE   NPROCS  PID  USER  COMMAND
2 4026533024 mnt         1   5195 root     bash
3 4026533025 uts         1   5195 root     bash
4 4026533026 ipc         1   5195 root     bash
5 4026533027 pid         1   5195 root     bash
6 4026533029 net         1   5195 root     bash
7 4026533105 cgroup      1   5195 root     bash
```

The question is then how the original image we pulled was built. For that purpose, the Dockerfile was created, which is a text representation of everything one would type to create a Docker Image. One side effect of how Docker Images are stored in memory is that they are layered by design, meaning they're composable.

Layers represent a change on the image, and each change is an instruction on the Dockerfile. With this in hand, we can build upon the Image we downloaded before by writing a Dockerfile.

Our Dockerfile contains two layers, one being the "base" layer from the internet (using the "FROM" directive), and the other replacing the "CMD" command, which before was running the bash shell and now will run the "echo Hello, Thesis!" command.

Listing 32 – Changing the default command on the upstream Ubuntu container image by using a Dockerfile.

```
1   FROM ubuntu
2
3   CMD ["echo", "Hello, thesis!"]
```

From this description we build the image using the docker build command, resulting as shown in Listing 33. Note that we have tagged the image as "hello-thesis", which is the same as giving an Image a human-readable name.

Listing 33 – AUsing the Docker CLI build command to build a container image from a Dockerfile description.

```
1   $ ubuntu docker build -t hello-thesis .
2   Sending build context to Docker daemon  2.048kB
3   Step 1/2 : FROM ubuntu
4   ---> cdb68b455a14
5   Step 2/2 : CMD ["echo", "Hello, thesis!"]
6   ---> Running in 9575a3194a48
7   Removing intermediate container 9575a3194a48
8   ---> 8f51e76ed4e3
9   Successfully built 8f51e76ed4e3
10  Successfully tagged hello-thesis:latest
```

If we run the Image as before, we get the desired result as shown in Listing 34.

Listing 34 – Running the newly built container with the modified command.

```
1   $ ubuntu docker run hello-thesis
2   Hello, thesis!
```

We can also easily share and deploy this "hello-thesis" Image everywhere that runs OCI images. With this negligible setup time and low overhead, it's clear why Container Images have become the de-facto standard tool for running software in the cloud and on development computers.

C.2 DOCKER COMPOSE

A tool that offers rudimentary orchestration capabilities is Docker Compose. It is used to define and run multiple containers, locally, on the same machine, and it lacks several features of full-fledged orchestrators being generally used as a wrapper around the Docker CLI.

To illustrate, we can orchestrate five replicas of our "hello-thesis" example from section C.1. For that, we define a YAML, seen in Listing 35, file that will be parse by the docker-compose python tool generate the proper *state* of containers. The result of the invocation section C.1 with docker compose up can be seen in Listing 36, which as expected creates 5 services each running our build of a custom container. Note that although Docker Compose has synchronization mechanisms, without specifying one the order which the services are brought up is non-deterministic.

Listing 35 – An YAML file to be used with the Docker Compose tool.

```
1  version: "3"
2  services:
3    hello-thesis-1:
4      image: hello-thesis
5    hello-thesis-2:
6      image: hello-thesis
7    hello-thesis-3:
8      image: hello-thesis
9    hello-thesis-4:
10     image: hello-thesis
11   hello-thesis-5:
12     image: hello-thesis
```

Listing 36 – Result of running docker compose up with the YAML file listed at Listing 35 as input.

```
1  Starting ubuntu_hello-thesis-2_1 ... done
2  Starting ubuntu_hello-thesis-5_1 ... done
3  Starting ubuntu_hello-thesis-3_1 ... done
4  Starting ubuntu_hello-thesis-4_1 ... done
5  Starting ubuntu_hello-thesis-1_1 ... done
6  Attaching to ubuntu_hello-thesis-2_1,
7      ubuntu_hello-thesis-4_1,
8      ubuntu_hello-thesis-5_1,
9      ubuntu_hello-thesis-1_1,
10     ubuntu_hello-  thesis-3_1
11 hello-thesis-2_1  | Hello, thesis!
12 hello-thesis-1_1  | Hello, thesis!
13 hello-thesis-3_1  | Hello, thesis!
```

```
14 hello-thesis-4_1  | Hello, thesis!
15 hello-thesis-5_1  | Hello, thesis!
16 ubuntu_hello-thesis-2_1 exited with code 0
17 ubuntu_hello-thesis-4_1 exited with code 0
18 ubuntu_hello-thesis-5_1 exited with code 0
19 ubuntu_hello-thesis-1_1 exited with code 0
20 ubuntu_hello-thesis-3_1 exited with code 0
```