UCLouvain

epl

École polytechnique de Louvain

Low-cost edge computing using upcycled smartphones

in collaboration with Swarn

Author: **Corentin Libert**
Supervisors: **Peter Van Roy, Tom Barbette**
Readers: **Jean-Brieuc Feron, Donatien Schmitz**
Academic year 2023–2024
Master [120] in Computer Science

Abstract

Smartphone users often replace their devices prematurely for newer models, contributing to the growing issue of waste electrical and electronic equipment (WEEE). Repurposing these devices to extend their life cycle by assigning them new roles can help mitigate this problem. This thesis explores the feasibility of creating a cluster using upcycled smartphones deployed with the K3S framework. We developed an image classification application capable of handling HTTP requests, utilizing the TensorFlow Lite and Crow frameworks. The devices, operating system, and framework were adapted for compatibility. Preliminary networking evaluations indicate that these devices can effectively be part of a cluster across various networking mediums. Performance evaluations of the image classification application running on this cluster indicate that performance scales effectively up to a medium-specific threshold with Ethernet and Wi-Fi 5.0GHz, confirming that upcycled devices are viable for a K3S cluster design aimed at low-cost edge computing.

Acknowledgements

First, I would like to thank my supervisors, Professor Peter Van Roy and Professor Tom Barbette, for their advice and guidance during this thesis, as well as for the resources that they have provided. I'm also grateful to Donatien Schmitz for agreeing to be part of my jury as a reader.

Secondly, I would like to acknowledge Swarn, and more specifically Jean-Brieuc Feron, for proposing this master's thesis and making it possible by providing the necessary material resources. I would also like to thank him and Nicolas Brusselmans for their technical assistance and interest in the project.

I would also like to express my gratitude to my family, whose encouragement and support in their ways have enabled me to reach where I am today.

Finally, and certainly not least, I would like to extend my deepest gratitude to my friends, whether part of the Réaumur group or not, for their help, support, and encouragement.

This document was written using ChatGPT[1], Gemini[2] and Grammarly[3] to improve expression, correct grammatical or syntactical errors, and refine the original content. These tools were solely used for reformulation and not for generating new content.

[1]https://chatgpt.com/
[2]https://gemini.google.com/
[3]https://www.grammarly.com/

Contents

Chapter 1

Introduction

Smartphones have become essential for an ever-growing number of people, serving various purposes such as communications, work, social media, entertainment and daily activities. As they become more and more present, their performance also keeps improving. Each smartphone generation comes with better capabilities and new functionalities, such as better communication capabilities, more powerful CPU and GPU chips, cameras and screens with better resolutions and always more sensors of various kinds. As a result, current phones have nothing more to do with their ancestors limited to communication purposes [1]. Instead, they are now small embedded computers with strong wired and wireless communication capabilities, as well as a broad range of sensing features.

Every year, phone manufacturer companies advertise new smartphone models. More than mere communication tools, today's smartphones are essential devices for everyday life, fashion accessories, and, for some, even an addiction [2]. Due to socio-economic factors, people tend to replace their phones prematurely for more recent models or due to a lack of software support [3], as shown by Figure 1.1.

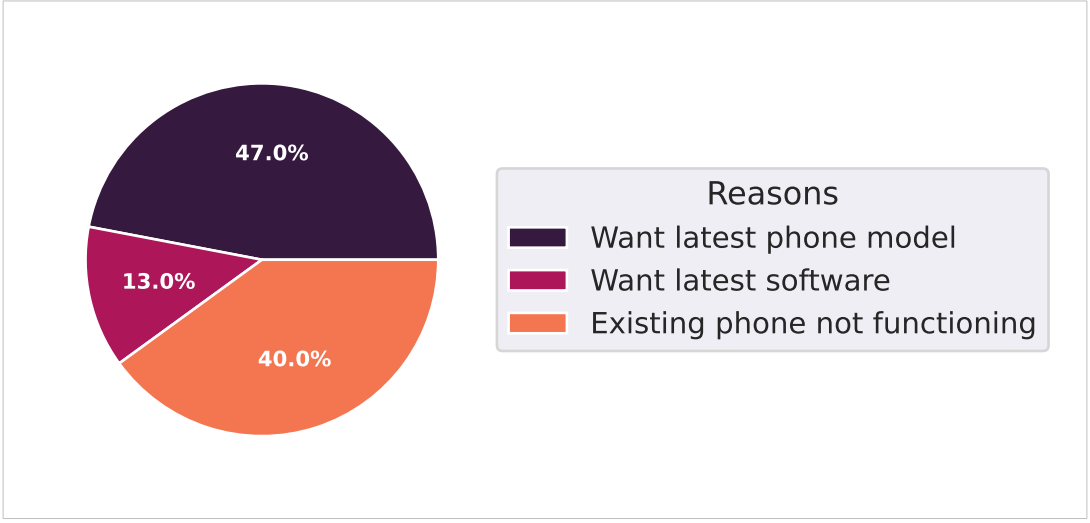

Figure 1.1: Reasons for smartphones replacement [3].

Even if they are still operational, those *electrical and electronic equipment (EEE)* become *waste electrical and electronic equipment (WEEE)*. As *WEEE* is part of the waste streams with the fastest growth [4], several ways to handle them exist, depending on the

state of the equipment.

Recycling is the most known one. Recycled devices will be dismantled to recover various materials such as polymers, metals and precious metals [5], ceasing to exist. These materials or components are then processed and reused for other purposes [6]. *Recycling* is ultimately the final solution for totally broken or outdated devices, working ones, even partially may have their life cycle extended through *refurbishment* and *repurposing*.

Refurbishment extends the device's life cycle by giving it a new life for the same purpose. If broken slightly, the device can be restored to a similar, the same or a better state than a new unit [6]. When a device is *refurbished*, it retains its original purpose and functionality, those of a smartphone. Refurbishment has its limits that can be at the physical level, the device is too broken, at the software level, the device is outdated and lacks support, or even at the psychological level, with psychological obsolescence and mental depreciation [7].

In this case, *repurposing* may still be a possibility. It extends the life cycle of the device by giving it a different purpose than the previous. Repurposing has the advantage of reusing devices for tasks where their eventual damages or outdated features don't have much importance.

In this master thesis, we focus on the repurposing of smartphones. Previous works have already demonstrated some of the possibilities and capabilities of repurposed smartphones in various application cases, as part of a cluster or not. For example, [8] demonstrates the potential of repurposed smartphones for elementary school education, emphasizing their multimedia capabilities which are argued to lead to better student achievement. They show that their evaluated device, the HTC Nexus One, satisfies the requirements for this use case, even after degradation due to previous utilisations. Focusing on the sensor capabilities of smartphones, [9] shows the feasibility of using upcycled smartphones as such for various campus application cases such as door indicators in real-time and a prediction system, Wi-Fi signal detectors or GPS balizes in a shuttle monitoring system. On a more scientific aspect, [10] proposes the utilization of LiDAR sensors from smartphone cameras to determine physical fluid properties for both biomedical and rheology applications. Also using the phone's camera, [11] demonstrates the possibility of detecting photons and muons, present in the air showers induced by ultra-high energy cosmic rays, using the CMOS sensors of the smartphone's cameras to observe such cosmic rays. By creating a cluster using a large number of smartphones, they palliate the small size and low efficiency of each sensor.
Previous works have shown the implementations and evaluations of clusters of recycled smartphones or other kinds of embedded devices, such as Raspberry PI, for various common application cases. Some focus on AI applications, such as the distributed training of deep learning models using smartphones [12] or image processing inference for agriculture purposes using Raspberry Pi [13]. Others discuss the use of such clusters for cloud computing applications [14] [15] [16]. There is also some work on data management using smartphone clusters [17].

This work focuses on the repurposing of smartphones in a cluster for low-cost edge applications. With the fast development of the Internet of Things (IoT), the number of smart devices and sensors is continually increasing. Placed on the edge of the network,

they collect local data that are then sent to data centers, owning the computing power, to be processed. In some cases, results are then sent back to the edge to be displayed to the user. However, the time needed for the transfer of the data and results may be too long for some application requirements, such as real-time applications. Edge computing appears as a solution to this problem, by deploying small-scale storage and computing installations closer to the edge [18].

In this work, we demonstrate the capabilities of repurposing smartphones as a low-cost cluster that uses the TensorFlow Lite framework to perform image processing at the edge of the network. The case study is inspired by smart surveillance cameras that can detect what is happening in real time [19]. We've built two cluster variants, one using only Fairphone 2, and the other using a Raspberry Pi 5 to manage the cluster. This work can be composed of two important parts, the creation of the image processing application using TensorFlow Flow that has been adapted to be compatible with our devices and the creation of the cluster.

This master thesis was elaborated and made possible by *Swarn*, which provided all the Fairphones 2.

Both the files required to set up the cluster and compile the TensorFlow Lite application, as well as the scripts and notebooks used for performance evaluation, are available on the GitHub repository, which can be found here:

`https://github.com/CorentinLibert/Cluster-of-upcycled-smartphones`

This manuscript is divided as follows:

Chapter 2: starts by giving background knowledges. It first goes deeper into some of the previously listed related works that focus on repurposing smartphones as clusters. It then describes the selected operating system (OS) and technologies for this work and why those have been selected. It characterizes the devices that will be used in this project and some adaptations that have been made.

Chapter 3: evaluates the networking performances of the devices that will be used in the cluster, i.e. the smartphones and the Raspberry Pi 5. This evaluation is crucial as networking capability is a key aspect of clustering performances.

Chapter 4: presents two designs and architectures of the cluster, one only composed of smartphones, while the other uses a Raspberry Pi 5 for cluster management.

Chapter 5: presents an adapted version of the image classification example given by TensorFlow Lite, to which HTTP requests handling capabilities have been added.

Chapter 6: evaluates the performance of the TensorFlow Lite application, as well as the scaling of those performances when the application is run on the cluster.

Chapter 7: concludes this thesis by summarizing the work done. Some future works ideas are also discussed.

Chapter 2

Background

This chapter gives some background information before going further. We begin by reviewing a selection of relevant related works that appear to be most pertinent to the direction chosen in this work. We then briefly describe the operating system and frameworks that have been used for this project. Finally, we shortly describe and give the specifications of every device, hardware and adapter that has been used to build the cluster.

2.1 Related work

The idea of repurposing outdated or partially damaged smartphones is not recent. Previous work has already proposed several approaches to reuse this kind of device for different purposes. We can observe two kinds of approaches.

On one hand, some approach proposes using smartphones individually. It does not mean that a smartphone may not be part of a larger system with other smartphones, but rather that it is considered a self-sufficient component capable of performing its role independently. For example [8] explores the potential of repurposed smartphones for educational and academic purposes, demonstrating that outdated smartphones still meet the requirements to be used by students. Through their multimedia capabilities, they offer a new range of possibilities that classic educational materials cannot. [9] proposes a different approach, focusing more on the sensor capabilities of the devices. This work proposes several applications using the sensing capabilities of smartphone sensors with a view to deployment on campus. In these applications, smartphones are used as GPS beacons to monitor shuttles, Wi-Fi signal detectors in study areas or even as door indicators to determine and monitor the availability of lab workspaces. In all these cases, smartphones are considered full-fledged components of the system, which fully accomplish their assigned mission.

On the other hand, some research proposes a different kind of approach, where smartphones are used as part of a collective system. Here, a single smartphone may lack the performance needed for specific tasks, so multiple smartphones are grouped together in a cluster to enhance their combined performance. As this master thesis follows the latter approach, we will concentrate on related works that also follow this method.

Cluster of smartphones for cloud purposes

[14] proposes to reuse discarded smartphones as an affordable cloud-computing alternative. As they warn of the lack of support for end-of-life Android devices, Jennifer Switzer and al. propose to replace the operating system with Ubuntu Touch, an Ubuntu open-source OS designed for mobile devices [20], allowing the device to be considered as a simple Linux machine. The design of their cluster consists of a phone bank, composed of repurposed smartphones that are responsible for performing general computational tasks. A Raspberry Pi is used to manage smartphones and to distribute the task among them. The cluster used a custom-made management system implemented as a Python-based server running on the Raspberry Pi. A corresponding management API working with HTTP requests and socketIO events has been developed for device and job management inside the cluster.

In their next work [15], they confirm their previous work by focusing on the energy consumption aspect of the cluster. They also improve their testing benchmark suite with some general computing tasks, like image resizing or k-nearest neighbors (KNN) classifier training. They conclude their work by reaffirming the viability of such a cluster as a cost and energy-efficient alternative to Function-as-a-Service (FaaS) provider for general-purpose tasks.

In their latest work [16], their main line of work was the environmental impact of their cluster, specifically its carbon footprint. In particular, they developed a new ideal carbon metric that could capture four points that they deemed as important. About the cluster design, they explain how to transform discarded phones into a general-purpose computing node, mainly by focusing on the cooling, networking and powering aspects. Finally, they continue along the same lines as their previous works by proposing a cloudlet that they compare in terms of performance, using the DeathStar benchmark suite, and in terms of carbon footprint, with their ideal metric, to old servers and current cloud provider solutions like AWS EC2 instances. They concluded that for some types of workloads, their cluster is cheaper and more carbon-efficient than traditional servers.

Cluster of smartphones for AI purposes

[12] demonstrates the performance of a huge cluster of 138 Samsung Galaxy S10+ to train deep neural networks. They kept the native Android OS of their devices. After a preliminary analysis, they observed that Wi-Fi 5.0 GHz has a reduced communication speed with the number of connected devices and that connecting more than 30 devices resulted in a high rate of disconnection. Therefore they decided to connect phones to three 48-port Ethernet switches, using muli-ports adapters supporting data transfer and a supply of power.

They implemented a program based on OpenMPI, a message passing interface (MPI) library, and Caffee, a deep learning library supporting OpenCL, to train deep neural networks. They showed that their cluster performance was comparable to that of general-purpose GPUs.

Instead of repurposed smartphones, [13] proposes to use a cluster made of Raspberry Pi 3 Model B to apply image processing to the field of agriculture. The design of their cluster is rather simple. It consists of a four-node cluster, one head node and three workers, connected to a public LAN through Ethernet. Raspberry operates under Raspian while the cluster is based on the MPI concept.

2.2 Operating system and frameworks

In this section, we briefly describe the selected operating system (OS) and frameworks that will be used in this project. We will also give some of their advantages and limitations.

2.2.1 postmarketOS

postmarketOS (pmOS) [21] is a free open-source operating system designed for smartphones and other mobile devices [22]. It allows running a Linux distribution on smartphones with a kernel as close as possible to the mainline Linux kernel, in comparison with Android which has a downstream kernel with a lot of patches [23]. Based on the lightweight distribution *Alpine Linux* [24], postmarketOS tries to keep a minimal version, lightweight and security-oriented. As Alpine Linux, postmarketOS is based on *musl libc*, a "lightweight, fast, simple and free" implementation of the C standard library [25] and *BusyBox*, a small executable combining tiny versions of common UNIX utilities [26].

The biggest advantage of postmarketOS is that it can be easily modified, increasing the ease of contributing to the project and making it evolve faster. Thanks to that, postmarketOS now supports, at different levels, more than 250 devices. In comparison, *Ubuntu Touch* [20] only support currently 26 devices. While most of the device ports are still in the testing phase, which is the case for the devices used in this project, it still allows the reuse of these devices to a certain point.

However, postmarketOS has also some disadvantages and limitations. The biggest one is the lack of compatibility, which is the result of its desire to be lightweight and its design based on Alpine Linux, using musl libc and BusyBox. While it has its advantages, musl libc is not as popular as other implementations of the C standard library, such as *glibc* [27]. Therefore to ensure compatibility, software and applications have to be compiled against musl libc (and the corresponding architecture) into a package that can then be installed. While most common Linux software has already a compiled package, it's not the case for all existing software and frameworks that may require still compilation, and even adaptation to ensure compatibility, from the developer.

Another drawback is the lack of support for legacy versions. Alpine's repository generally contains only the latest stable versions of packages, making it difficult to use older versions and sometimes leading to compatibility issues between packages. postmarketOS itself also decreases the support for previous versions by forcing the installation of new kernel versions once they have been upgraded. This makes it challenging, if not impossible, to maintain multiple devices running the same Linux kernel version if it has been updated between different flashing cycles.

2.2.2 TensorFlow Lite

TensorFlow [28] is a free end-to-end open-source machine learning platform developed by Google. It can be used for a broad number of tasks but is specialized in the training and inference of deep neural networks. Currently, like PyTorch, TensorFlow is one of the most popular frameworks in ML. Mainly written in `C++` and `Python`, it provides a stable Python API, APIs without stability promises for `C++`, `Java` and `JavaScirpt`, as well as community supported package for others languages. Since version 1.14, TensorFlow only supports 64 bits systems, making 32 bits only able to run outdated versions of the framework.

Fortunately, since a few years ago, a lightweight version of TensorFlow has made its appearance, *TensorFlow Lite* [29]. Unlike the complete version of TensorFlow, the Lite version only supports inference, using adapted versions of trained TensorFlow models. TensorFlow Lite is designed for embedded devices, such as smartphones and Raspberry Pi, which is compatible with both 32 bits and 64 bits systems. While some compiled packages exist for common devices, like Raspberry Pi, TensorFlow Lite comes with building scripts for a broad range of devices. It can be built for `Android`, `iOS` and other `ARM` devices such as Raspberry Pi.

2.2.3 Crow

Crow is a lightweight micro-framework for the web written in `C++`. As it's based on *Flask*, a well-known similar framework written in Python, for its routing mechanism, it makes it easy to use. *Crow* allows the creation of HTTP and WebSocket web services with ease. While particularly fast, Crow is still less known than other similar `C++` or `non-C++` frameworks that may offer a broader range of functionalities and support.

2.2.4 K3S

K3S is a lightweight certified Kubernetes distribution built for IoT & Edge computing. It is perfect for edge and is optimized for ARM processors, including both ARM64 and ARMv7. It is thus compatible with the ARMv7l processor of the Fairphone 2, as well as the ARM64 of the Raspberry Pi 5. *K3S* allows to create and manage a fleet of nodes on which containerized applications can be deployed.

As for Kubernetes, machines, either physical or virtual, being part of the cluster are called nodes. A node is assigned one of two possible roles, either it becomes a **server node** or it becomes a **agent nodes** (called a **worker node** in Kubernetes). **Server nodes** are responsible for the cluster management, i.e. manage other nodes, deployments, services, etc., they have control-plane and datastore components managed by K3S. On their part, **agent nodes** do not have such components and are designed to host pods that run containers with some application. Pods are considered to be the smallest deployable unit that we can create and manage. By default on *K3S*, a **server node** also runs an agent node, meaning that it can host pods running applications.

Compared to *Kubernetes*, *K3S* is simpler to install. It provides a single script available online that can be run using a `curl` command. The script installs a *K3S* and configures a **server node** on the device. Once installed, a token can be retrieved from the server node and used as a flag to install K3S and configure an **agent node** that will join the cluster through the server.

Once a cluster is set up, applications can deployed. A **deployment** manages a set of pods that run containerized applications. They may be configured to run only on certain nodes or not. K3S allows to rescale of a deployment by creating new pods or destroying existing pods. A **service** can be linked to a set of pods, to offer networking service, for example, load balancing.

One reason K3s is easy to use is its configuration; it includes default plugins that can be replaced with others if needed. The four main plugins used are:

- *Flannel*, the container network interface (CNI).

- *Traefik*, the Ingress controller and for routing the HTTP/HTTPS traffic to the appropriate service with the cluster.
- *ServiceLB*, as the load balancer for `LoadBalancer` service.
- *Containerd*, as the container manager.

2.3 Machines and adapters specifications

In this section, we will list and give every technical specification relevant to this work of all machines and adapters that will be part of the cluster. We will also briefly explain their purpose and define some equivalent terms by which they will be called in the rest of this document.

2.3.1 Fairphone 2

The *Fairphone 2* is the second model of smartphone manufactured and released in 2015 by the Dutch company *Fairphone* [30]. With a more ecological approach, it is designed to be modular, allowing for easy disassembly and replacement of defective parts.

The relevant technical specifications of the Fairphone 2 are the following [31]:

- **System-on-chip:** Snapdragon 801, Qualcomm MSM8974AB-AB
- **CPU:** Quad-Core 2.26 GHz Qualcomm
- **Processor architecture:** armv7l
- **GPU:** Qualcomm Adreno 330 GPU 578 MHz, 2 GB LPDDR3
- **RAM:** 2 GB LPDDR3
- **Flash storage:** 32 GB eMMC5
- **Connectivity:**
  - **Wi-Fi:** IEEE 802.11 b/g/n/ac, 2.4GHz and 5.0GHz, with 1x1 antenna
  - **USB port:** USB Port: Micro-B 2.0 with OTG support
- **Battery and Power:** Removable Battery Pack: 2420 mAh at 3.8V (9.2 Wh), Lithium-Ion

This smartphone model is the one that will be used in this work as a reference to demonstrate the possibility of repurposing upcycled smartphones.

In the rest of this document, the terms *smartphone*, *Fairphone 2* and *device* may be used equivalently.

Setup, configuration and power supply

Fairphones 2 used in this work have been provided by *Swarn* [32].

For convenience reasons, we kept most of the default configuration and adapted it when needed. Therefore, the phones operate under *postmarketOS*, with version `6.9.1` of the Linux kernel. The postmarketOS port for this phone model is still a port under testing developed by the community. Therefore, only a part of the device functionalities are completely functional, while others are partial or not working at all [33].

We also used the provided adapters for the Ethernet connection of the phones. This is done by plugging a micro-USB (male) to USB (female) adapter to the micro-USB port of the Fairphone 2, then connecting it to a USB (male) to Ethernet (female) adapter. An Ethernet cable can then connect the adapter to any device having an Ethernet port. Figure 2.1 shows the setup on one smartphone.

Figure 2.1: Photo representing the micro-USB to Ethernet connection setup using two adapters.

Note that the USB (male) to Ethernet (female) is only compatible with USB 2.0 and is limited to 100 Mbit/s.

As the micro-USB port of the smartphones may be used for Ethernet communication, some adjustments have been made to power the phone without using it. The solution implemented by *Swarn* is to remove the phone's battery and to connect the internal cable of a USB cable to the battery connectors. A 3D printed support allows to keep the pins in contact with the battery connectors. A hole has been made in the back cover to allow the cable to pass. The other side of the cable is a USB (male) connector. The power supply setup is represented in Figure 2.2.

Figure 2.2: Photo representing the power supply adjustments made on the Fairphone 2.

2.3.2 Raspberry Pi 5

The *Raspberry Pi 5* is the last model of Raspberry released on October 2023. Well-known, Raspberry Pis are small embedded computers at a relatively low price. It is designed for a large range of tasks, such as smart home automation, IoT applications, low-power servers, ...

The relevant technical specifications of the Raspberry Pi 5 are the following [34]:

- **Processor:** Broadcom BCM2712 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU, with Cryptographic Extension, 512KB per-core L2 caches, and a 2MB shared L3 cache
- **GPU:** VideoCore VII GPU, supporting OpenGL ES 3.1, Vulkan 1.2
- **RAM:** 8 GB LPDDR4X-4267 SDRAM
- **Storage:** 128 GB micro SD card, U3 (UHS Speed Class), V30 (Video Speed Class), A2 (Application Performance Class)
- **Connectivity:**
  - **Wi-Fi:** IEEE Dual-band 802.11 b/g/n/ac Wi-Fi (2.4GHz and 5.0GHz), with 1x1 antenna
  - **Ethernet:** Gigabit Ethernet, with PoE+ support (requires separate PoE+ HAT)
  - **USB:** 2 x USB 3.0 ports, supporting simultaneous 5Gbps operation; 2 x USB 2.0 ports
- **Battery and Power:** 5V/5A DC power via USB-C, with Power Delivery support

In this work, the Raspberry Pi 5 operates under `Debian GNU/Linux 12 (bookworm)` OS with the Linux kernel version `6.6.31-v8-16k+`.

This model of Raspberry has better specifications than the Fairphone 2, more precisely, it has an Ethernet port, four times more RAM and a more performant CPU. Its purpose is to be used as a server node of the cluster, where the Fairphone 2 may be lacking in performance. As we only have a single smartphone model available for this work, we can see the Raspberry Pi 5 as a temporary necessity that could be replaced by a more recent smartphone model later.

In the rest of this document, the terms *Raspberry Pi 5* and *device* may be used equivalently.

2.3.3 Mikrotik cAP ac

The *Mikrotik cAP ac* is a powerful wireless access point with routing capability manufactured by *Mikrotik* [35]. It supports dual-band wireless radio for both 2.4GHz and 5.0GHz, with support for 802.11b/g/n protocols and 802.11a/n/ac protocols. Therefore, the Wi-Fi 2.4GHz is of the Wi-Fi 4 generation while Wi-Fi 5.0GHz is of the Wi-Fi 5 generation.

It also possesses two Ethernet ports with 10/100/1000 support. The power is supplied through one of the Ethernet ports.

In the rest of this document, the terms *Mikrotik cAP ac* and *router* may be used equivalently.

2.3.4 HP 2530-48G J9775A Switch

The *HP 2530-48G J9775A Switch* is a layer 2 switch with 48 ports manufactured by HP. Each port has 10/100/1000 support.

In the rest of this document, the terms *HP 2530-48G J9775A Switch* and *switch* may be used equivalently.

2.3.5 HP Probook 450 G6

The *HP Probook 450 G6* is a laptop manufactured by *HP*. The relevant technical specifications of the model used are the following [36]:

- **Processor:** Intel Core i5-8265U processor. 1.6 GHz base frequency, up to 3.9 GHz with Intel Turbo Boost Technology. 6 MB L3 cache and 4 cores
- **GPU:** Intel UHD Graphics 620
- **RAM:** 2 x 16 GB SODIMM DDR4 Synchronous 2667 MHz
- **Connectivity:**
  - **Wireless LAN (WLAN):**
    - Intel Dual Band Wireless-AC 9560 802.11ac (2x2) WLAN and Bluetooth 5 Combo, non-vPro
    - Realtek RTL8821CE 802.11ac (1x1) WLAN and Bluetooth 4.2 Combo
    - Realtek RTL8822BE 802.11ac (2x2) WLAN and Bluetooth 4.2 Combo
  - **Ethernet:** Realtek RTL8111HSH-CG 10/100/1000 GbE NIC

In this work, this laptop will act as a server. Therefore, in the rest of this document, the terms *HP Probook 450 G6*, *laptop* and *server* may be used equivalently.

Chapter 3

Preliminary network evaluation

In this chapter, we will show and analyze the results of several networking experiments that have been performed to determine the global networking capabilities of the devices that will be used in the cluster, i.e. Fairphone 2 and Raspberry Pi 5. The network communication capabilities are a key component in a cluster, which is based on communication between nodes. Therefore performing a preliminary analysis of the devices that will be part of it is a necessity and will allow us to better analyze further evaluations of the cluster performance. The rest of this chapter is sctuctured as follows. First, we present the metrics on which we will focus during the experiments. We then describe the measurement scripts and the default settings common to both experiments. Afterward, we describe each experiment, explaining their purpose, showing their respective topology setup and giving the results of it and their interpretation. Finally, we give a short conclusion of the networking capabilities of smartphones based on the analyzed results of both experiments.

3.1 Experiment metrics

In the following experiments, we are interested in the bandwidth, more particularly the goodput, and the latency:

- *Goodput:* The goodput corresponds to the average amount of data correctly transmitted per unit of time on the application-level of a network communication. In contrast to the throughput, goodput only takes into account useful data for the application, excluding things like protocol overhead, retransmission of packets due to loss, and other layer control messages. The higher the goodput, the better. As we work with devices that have common to poor networking performance, the unit of measure will be *Mbit/s*. Through this measure, we aim to determine how much data our device will later be able to send in the cluster. We determined the goodput of communications using the software `iperf3` [37] that creates a client-server communication between two devices and measures its goodput.

- *Latency:* The latency of a network communication corresponds to the time taken for data packets to travel from one place to another. It is a synonym for the delay of the transmission. The lower the latency, the better. We use *milliseconds (ms)* as the unit for the latency. Through this measure, we aim to analyze the average delay of communications between the devices that will later compose the cluster. We determine the latency using the command `ping` [38].

3.2 Testbed

We've performed two experiments to determine the previously defined network capabilities of the devices over several networking mediums. This section describes the measurement script and default settings that are common to both experiments. As the topologies are different for each experiment, they will be described in their respective experiment sections.

3.2.1 Measurement scripts

We implemented two bash scripts that allow us to perform goodput and latency measurements respectively, on several remote devices and at the same time. Both scripts consist of creating a remote workspace on the given devices through SSH, performing the measure and then retrieving the results to the host machine with the `scp` command.

The goodput script performs measurements by setting up client-server communication between pairs of sender-receiver devices, given as arguments, using `iperf3`. The script's arguments allow the configuration of the duration of the `iperf3` communication, as well as the number of first results that should be omitted.

The latency script works with the `ping` command, pinging from the sender device to the receiver device. Furthermore, the latency script allows the simulation of network traffic between the pairs of devices using `iperf3`. Similarly to the goodput script, the latency script arguments allow to determine the number of measures taken by the `ping` command, with an interval of one second between pings, as well as the number of first results that should be omitted.

3.2.2 Terms definitions and default settings

Both experiments use the same machines and adapters to form their topology. Hereby, we remind you of some equivalent terms for each machine used in the rest of this chapter. These have already been defined in section 2.3 where their relevant specifications were also given.

- **Device:** corresponds to either Fairphone 2 or Raspberry Pi 5 (or both).
- **Server:** corresponds to the *HP Probook 450 G6* that will acts as one.
- **Switch:** corresponds to the HP 2530-48G J9775A Switch.
- **Router:** corresponds to the Mikrotik cAP AC.

Each experiment measures the goodput and latency performance over the communication(s) in both ways, over several networking mediums, using the scripts described previously. The selected networking mediums and their configuration are the following:

- **Wi-Fi 2.4GHz**: The wireless medium Wi-Fi 2.4GHz with 20/40MHz channel width, only authorizing the Wi-Fi protocol 802.11n (Wi-Fi 4).
- **Wi-Fi 5.0GHz**: The wireless medium Wi-Fi 5.0GHz with 20/40/80MHz channel width, only authorizing the Wi-Fi protocol 802.11ac (Wi-Fi 5).

- **Ethernet**: The wired medium Ethernet. Note that adapters are used to adapt the micro-USB port of the smartphones to the Ethernet port of the switch.

Each run of the goodput or latency script lasts for 15 seconds. The interval between measures is one second, resulting in a total of 15 measures per run. The 5-first results of each run, corresponding to the connection setup time, are omitted.

3.3 Experiment 1: Maximal networking performance of the devices

In this experiment, we aim to determine the maximal networking performance, i.e. goodput and latency, that are reachable by each device, i.e. Fairphone 2 and Raspberry Pi 5, on the selected networking mediums. Knowing such performance is crucial for further experiments and analysis.

The experiment consists of measuring the goodput and the latency between the measured device and the server, over each medium. These measures are done both ways, i.e. with the device being the sender and then the receiver.

We measure the performance between the device and a server because our router does not allow us to easily measure the good output between itself and the device. However, to minimize the impact of the transmission between the router and the server, we directly connected the latter to the router through an Ethernet cable. Therefore, as this part of the transmission is estimated negligible, we've decided to take it as part of the performance of the measured devices.

Topology

The topology for this experiment is shown in Figure 3.1.

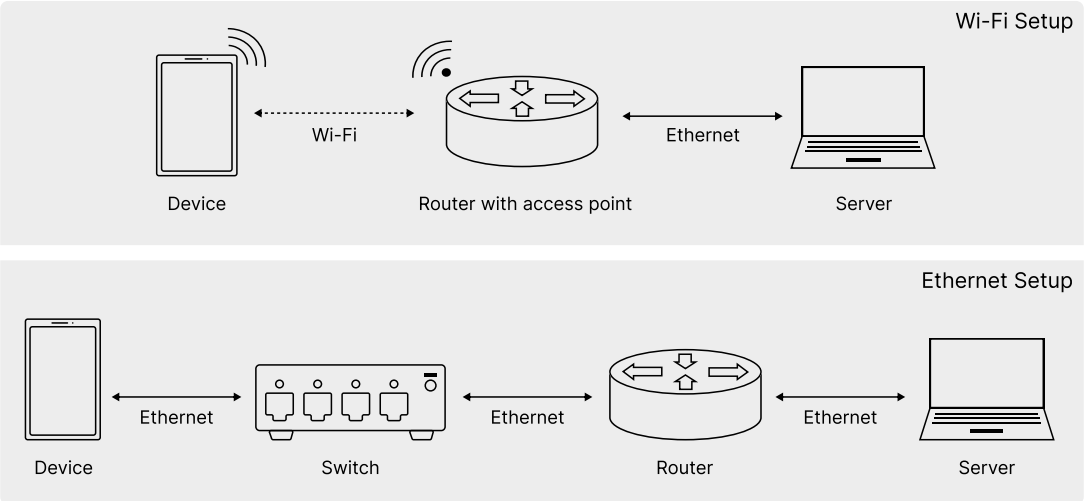

Figure 3.1: Experimental wired and wireless setups to determine maximal networking performance of measured devices, over different networking mediums.

The topology varies depending on whether the medium is wireless or wired. In both cases, the server is directly connected to the router via an Ethernet connection. Given

that Ethernet offers significantly better performance compared to other mediums, and based on the hardware specifications of the devices in the topology, we assume that the communication between the router and the server has a negligible impact. We also assume that no machines in the topology, other than the devices being tested, would act as a bottleneck in the measurements.

For the wireless topology variant, the device is directly connected to the router through the access point of the latter. For the wired topology variant, the device is connected to the switch, connected to the router, through Ethernet. The adapters ensure compatibility between the micro-USB port of the Fairphone 2 and the Ethernet port of the switch (see section 2.3.1).

Based on their specifications (as seen in section 2.3), neither the router, the switch nor the server should act as a bottleneck and therefore should not limit the performance of the measured devices.

Results and interpretation

We will now present and analyze the maximal network performance of the devices for each medium. We will regroup the results by goodput and latency. We will do a per-medium analysis before ending by giving a small conclusion on the results of this experiment.

Goodput

Starting with the wireless medium, Figure 3.2 shows the results of the goodput for Wi-Fi 2.4GHz.

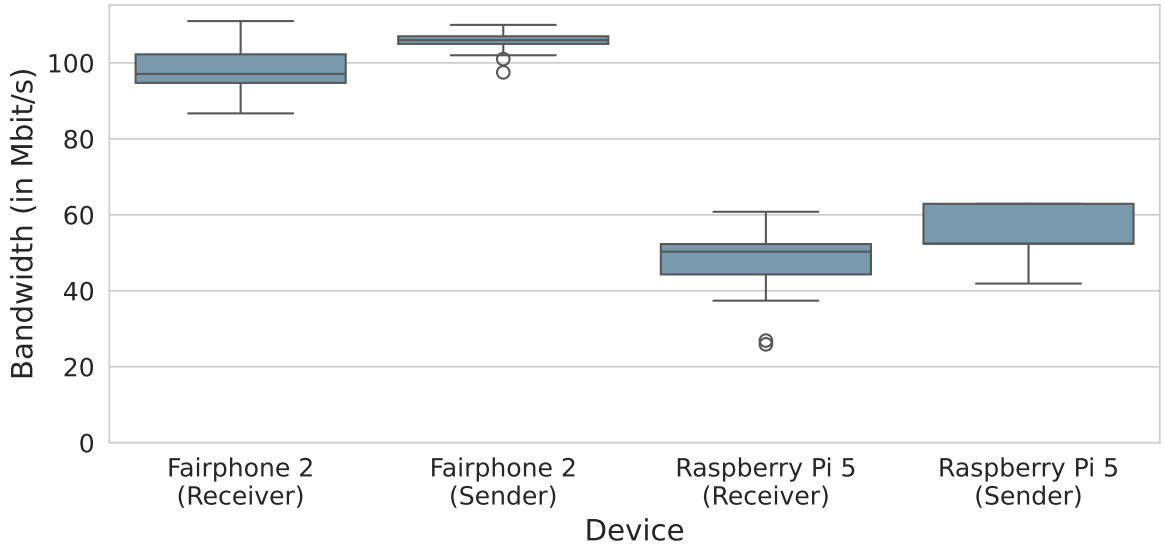

Figure 3.2: Maximal goodput for Fairphone 2 and Raspberry Pi 5 over Wi-Fi 2.4GHz with 20/40MHz channel width using the 802.11n protocol.

The first observation is that the Fairphone 2 outperforms the Raspberry Pi 5. The latter being more recent than the smartphone, we would've expected better performance from the it. After more analysis, the reason is that the Raspberry Pi 5 does not support 40MHz channel width for Wi-Fi 2.4GHz. This issue was already reported on previous

Raspberry Pi versions [39][40]. More precisely, since the chip is compatible with 40MHz, it seems to be a problem with firmware, not allowing *channel bonding*.

We also observe that the goodput is quite stable, with still some outliers due to the wireless nature of the medium. Both devices seem to perform a little better when sending than receiving.

For Wi-Fi 5.0Ghz, we can see on Figure 3.3 that the Raspberry Pi 5 outperforms the Fairphone 2, as we would expect.

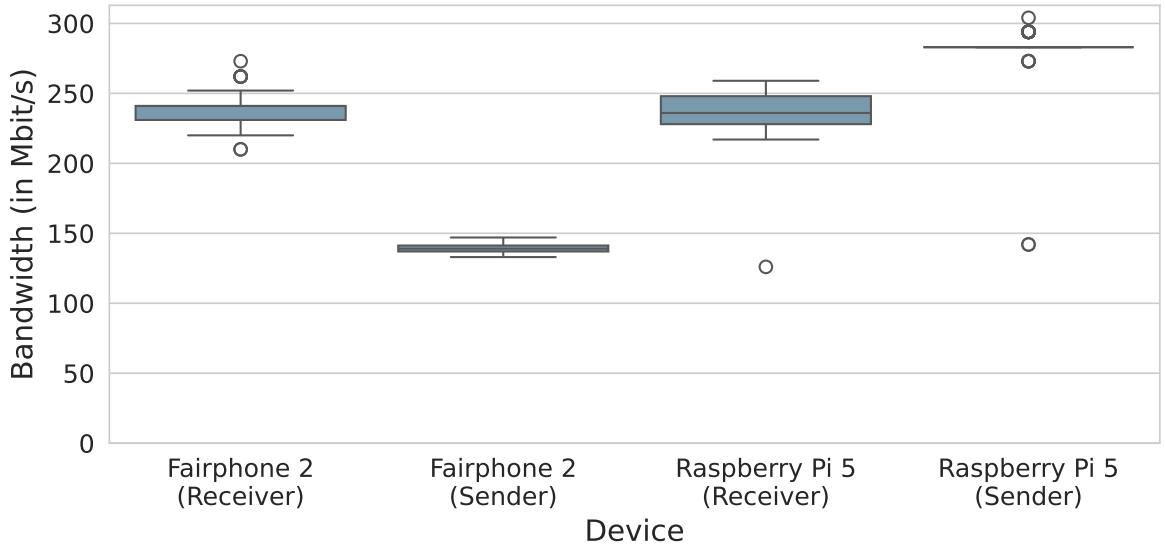

Figure 3.3: Maximal goodput for Fairphone 2 and Raspberry Pi 5 over Wi-Fi 5.0GHz with 20/40/80MHz channel width using the 802.11ac protocol.

As for Wi-Fi 2.4GHz, the goodput of the Raspberry Pi 5 is still slightly better when sending than when receiving. However, this is not the case for the Fairphone 2 whose goodput is around 100 Mbit/s higher when receiving than when sending. We explain this difference by the age of the device and suppose that it is a manufacturer device design since smartphones are more likely to receive than send on Wi-Fi. While the goodput difference is particularly visible between senders, both devices perform similarly when receiving. The goodput is also quite stable, with still some outliers, as for the Wi-Fi 2.4GHz.

Finally, for Ethernet, we observe an important difference in goodput between both devices, as shown in Figure 3.4. Note the broken y-axis on the figure.

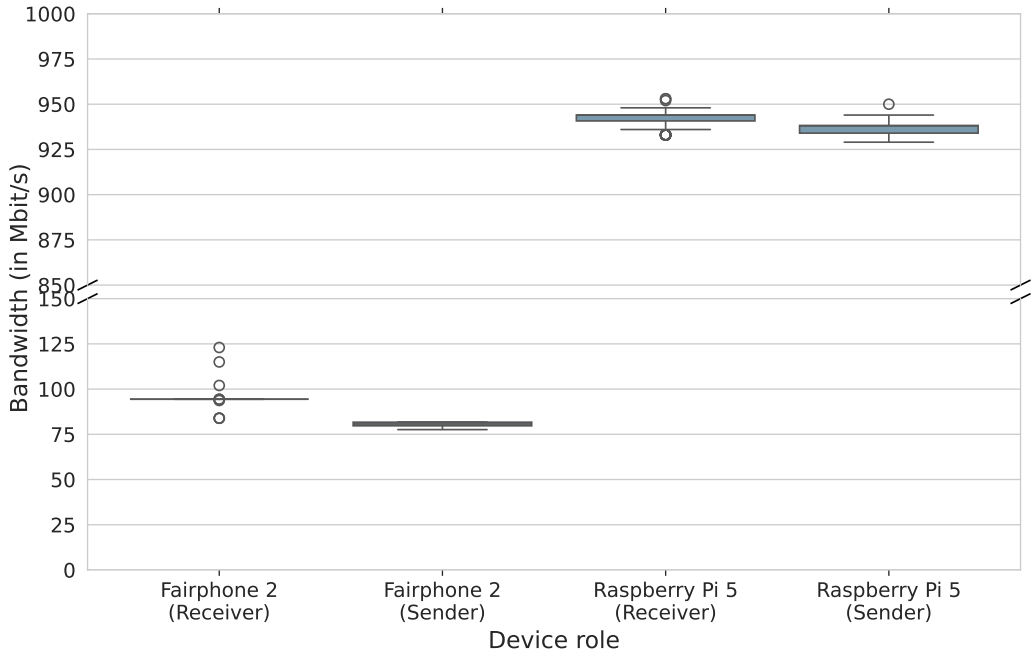

Figure 3.4: Maximal goodput for Fairphone 2 and Raspberry Pi 5 over Ethernet. Micro-USB to Ethernet adapters have been used for Fairphone 2.

While the Raspberry Pi 5 nearly achieves 1 Gbit/s, with around 940 Mbit/s for both sending and receiving, the Fairphone 2 does not exceed 100 Mbit/s in either direction. This disparity can be attributed to the hardware differences: the Raspberry Pi 5 has a built-in Ethernet port, while the Fairphone 2 relies on a micro-USB 2.0 Type B connection. The theoretical maximum throughput of USB 2.0 is 480 Mbit/s, but in practice, this is much lower and depends on the quality of the USB cable used.

Additional measurements indicated that when connecting the smartphone directly to the server via Ethernet over USB, without a router, we achieved 175 Mbit/s for sending and 140 Mbit/s for receiving. The difference in performance with the values in Figure 3.4 are explained by the USB-Ethernet adapter that limits the throughput to 100 Mbit/s (see section 2.3.1). Consequently, the goodput was lower than the theoretical throughput. Compared to the wireless mediums, the Ethernet medium has even more stability. However, some outliers are still present and some random disconnections from the medium were rarely observed.

Latency

Starting again with the wireless mediums, Figure 3.5 and Figure 3.6 show the latency for Wi-Fi 2.4GHz and Wi-Fi 5.0GHz respectively.

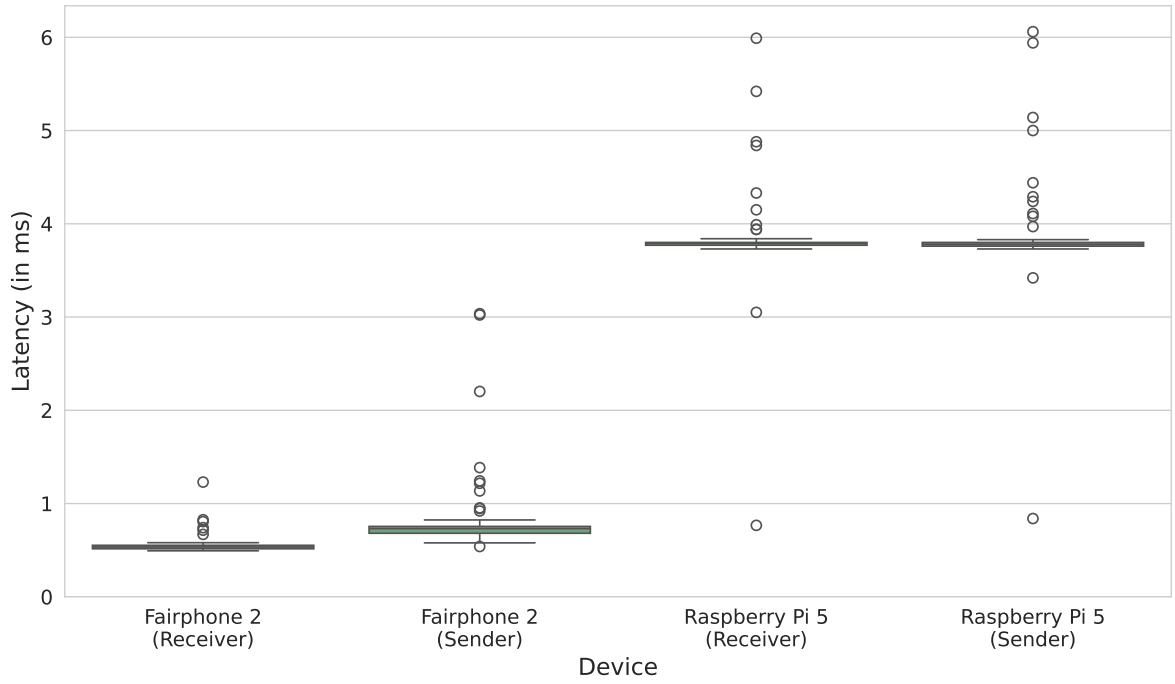

Figure 3.5: Maximal latency for Fairphone 2 and Raspberry Pi 5 over Wi-Fi 2.4GHz with 20/40MHz channel width using the 802.11n protocol.

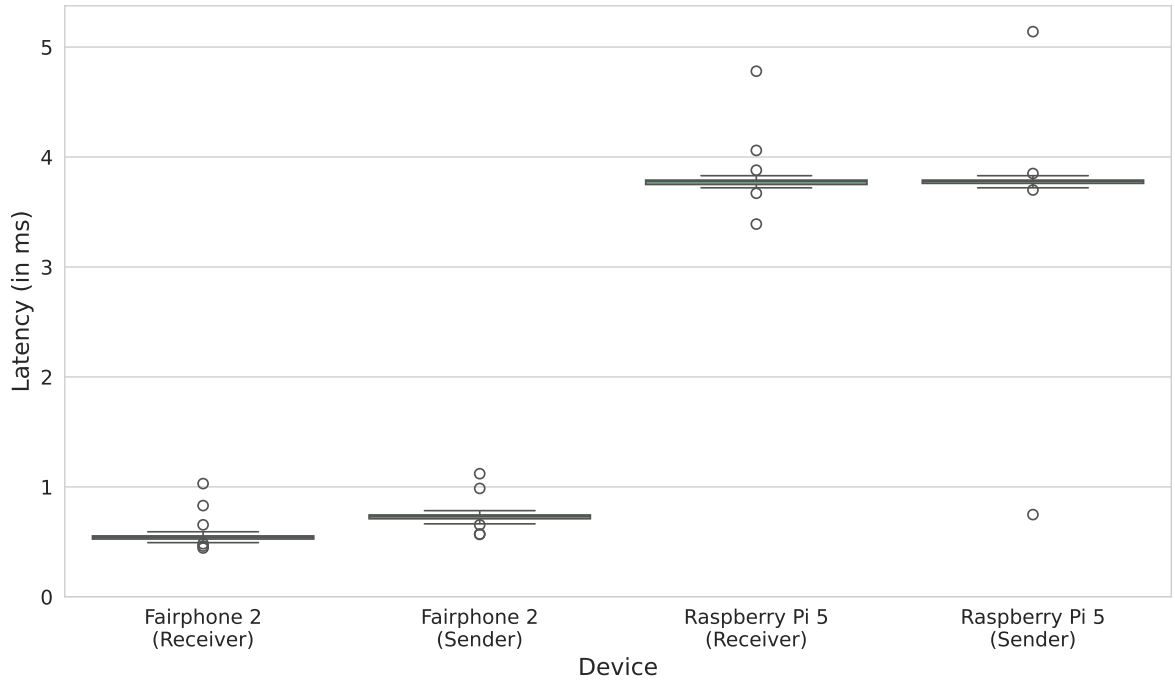

Figure 3.6: Maximal latency for Fairphone 2 and Raspberry Pi 5 over Wi-Fi 5.0GHz with 20/40/80MHz channel width using the 802.11ac protocol.

Both results are quite similar. We observe that the Fairphone 2 has a quite low latency of less than one millisecond, for both sending and receiving, while the latency of

the Raspberry Pi 5 is greater, near the four milliseconds. These results show that the Fairphone 2 may be a better choice for applications requiring a shorter response time. While the variance of the results is low, we still observe some outliers that can be explained by interferences, common in wireless mediums.

Figure 3.7 shows that the Ethernet medium gives different results.

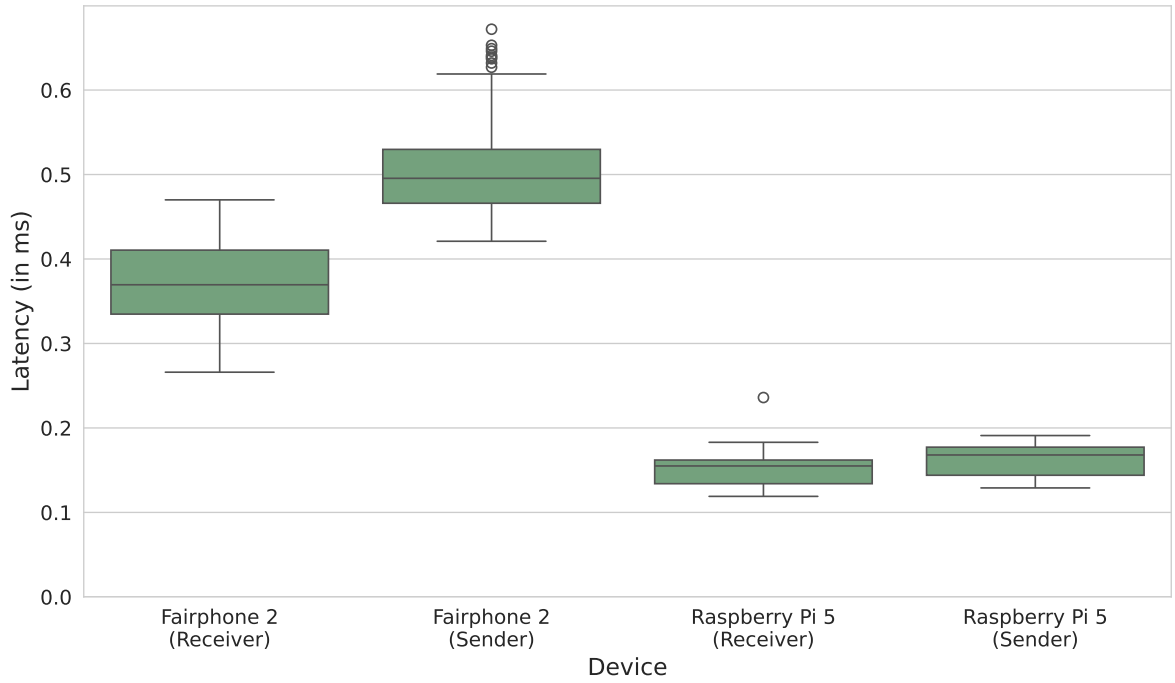

Figure 3.7: Maximal latency for Fairphone 2 and Raspberry Pi 5 over Ethernet. Micro-USB to Ethernet adapters have been used for Fairphone 2.

In comparison with the wireless mediums, for the Ethernet medium, the Raspberry Pi 5 outperforms the Fairphone 2. While the latter has a latency slightly lower than before, around 0.4 to 0.5 milliseconds, the Raspberry drops from around four milliseconds to less than 0.2 milliseconds. Globally latency is stable with few outliers, which is common for wired mediums.

Conclusion

These results demonstrate that the wireless networking performance of old smartphones may be comparable to or even better than those of more recent embedded devices, be it in latency or goodput. While behind sometimes, smartphones still are a viable alternative from a networking point of view for IoT applications.

In the case of wired networking performance, Fairphone 2 is not a match at all for Raspberry Pi 5. While the difference is small when looking at the latency, the difference in goodput is way too important. Fairphone 2 may still be an interesting option for IoT applications with a low amount of data transferred. However, it is important to note that the current performance is limited by both the adapters and the maximal throughput of the USB 2.0 standards. With better adapters, not limited to 100 Mbit/s, and smartphones supporting the USB 3.0 standard, with a maximal theoretical throughput of about 1 Gbit/s, we may reach similar wired networking performance to the Raspberry Pi 5.

3.4 Experiment 2: Evolution of the networking performance of the devices for an increasing number of devices

In this experiment, we observe the evolution of the networking performance of devices between them for an increasing number of active devices connected to the medium. By *active devices* we mean a device that sends data to or receives data from a connection with another device. We limit the number of connections to one by device, where it is either the sender or the receiver. We start the experiment with two devices, with one connection between them, to end with 16 active devices, with a total of 8 connections. We increase the number of active devices by two so that the number of connections increases by one.

The final purpose of this experiment is to determine how the performance of the communication between devices evolves when their number increase, in the prevision of them being part of a cluster later. Therefore, we analyze three types of communication that would be present in it:

- From Fairphone 2 (sender) to Fairphone 2 (receiver)
- From Fairphone 2 (sender) to Raspberry Pi 5 (receiver)
- From Raspberry Pi 5 (sender) to Fairphone 2 (receiver)

For the latency, we've performed two variants of the experiment. The first one is the classic one, we measure the latency between active devices that are having a workload generated by `iperf3`. In the second one, the latency is measured with *idle devices*, which are devices connected to the medium but without any workload being generated. Each variation of this experiment has its purpose. While the first one shows latency when we simulate a working cluster, the second one demonstrates the possible impact on the latency of the number of idle devices simply connected to the same medium.

Topology

The topology for this experiment is shown in Figure 3.8.

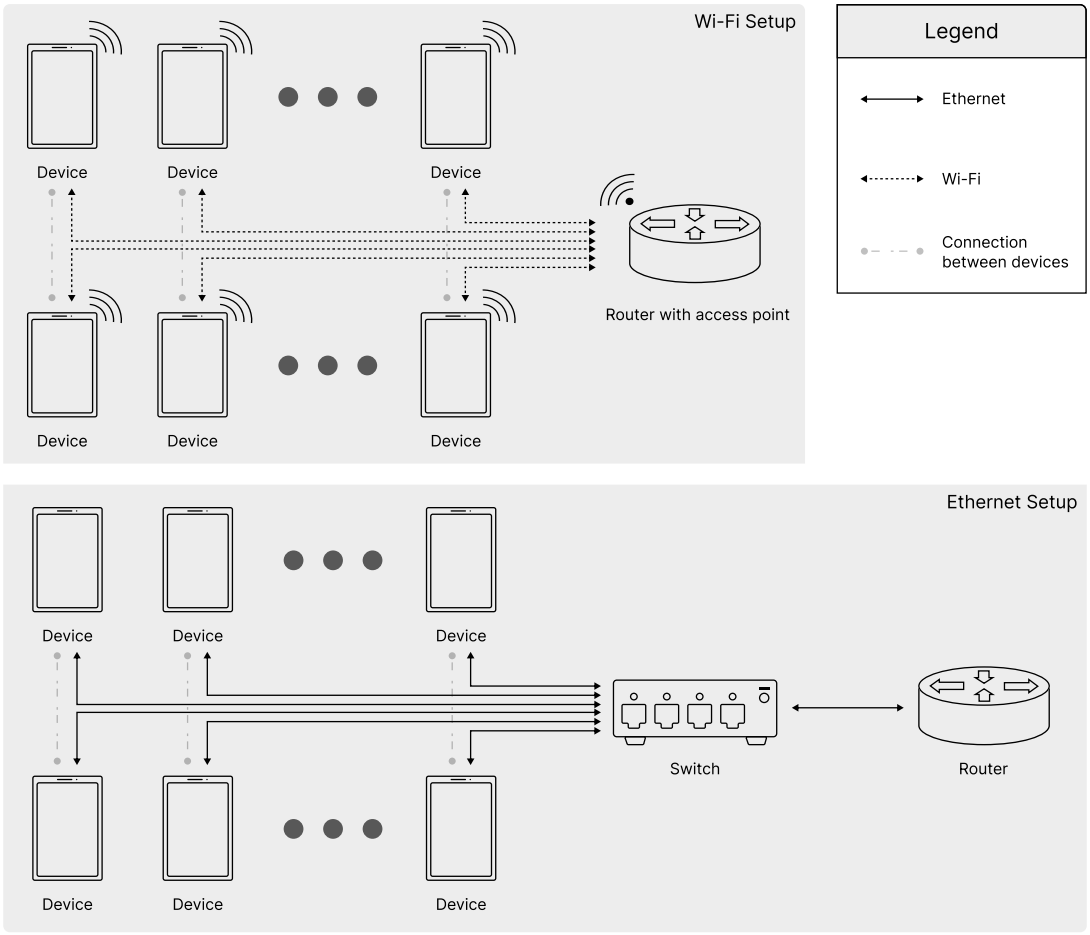

Figure 3.8: Experimental wired and wireless setups to determine the evolution of networking performance of devices for an increasing number of connected devices with active connections, over different networking mediums.

As for the first experiment, depending on whether the medium is wireless or not the topology differs. The topology's variants are similar to the ones of the first experiment, except for the server that is not used and thus present anymore. Notice in the figure that annotations between devices were added. These represent the connections created during the experiment, at the transport layer level, between pairs of sender-receiver devices.

Results and interpretation

We will now present and analyze the networking performance of the devices for an increasing number of them, for each medium. Results will be regrouped by goodput and latency and we will do a per-medium analysis. We then terminate with a small conclusive interpretation of the results.

Goodput

The goodput for wireless medium decreases with an increasing number of active devices connected to the medium. Figure 3.9 and Figure 3.10 have a similar a similar hyperbolic curve form.

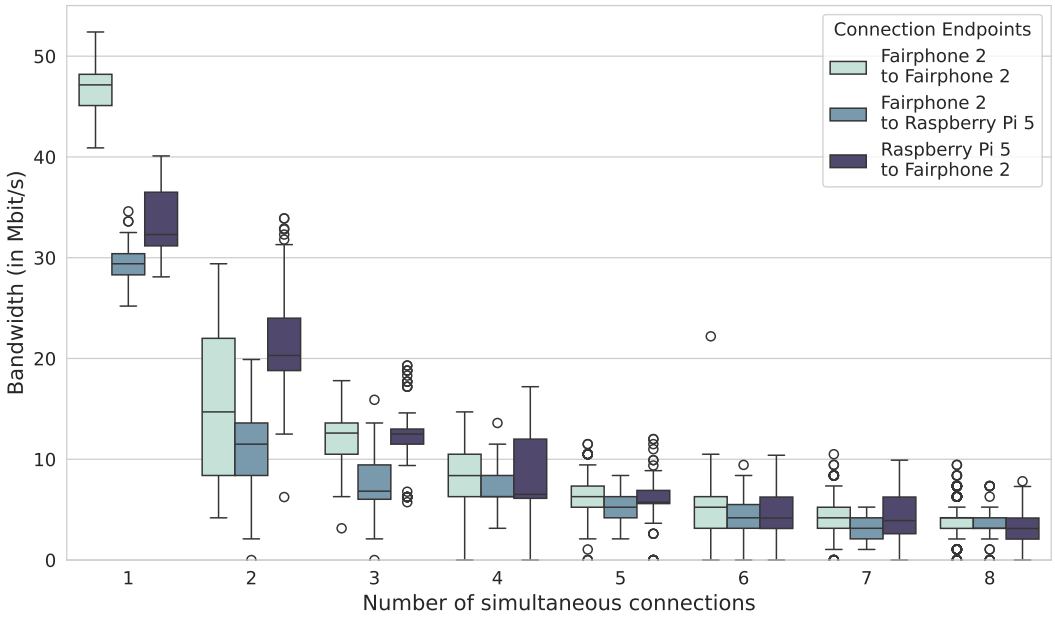

Figure 3.9: Evolution of the goodput per connection between different pairs of active devices for an increasing number of active devices, with a single connection between two active devices, over Wi-Fi 2.4GHz with 20/40 channel width using the 802.11n protocol.

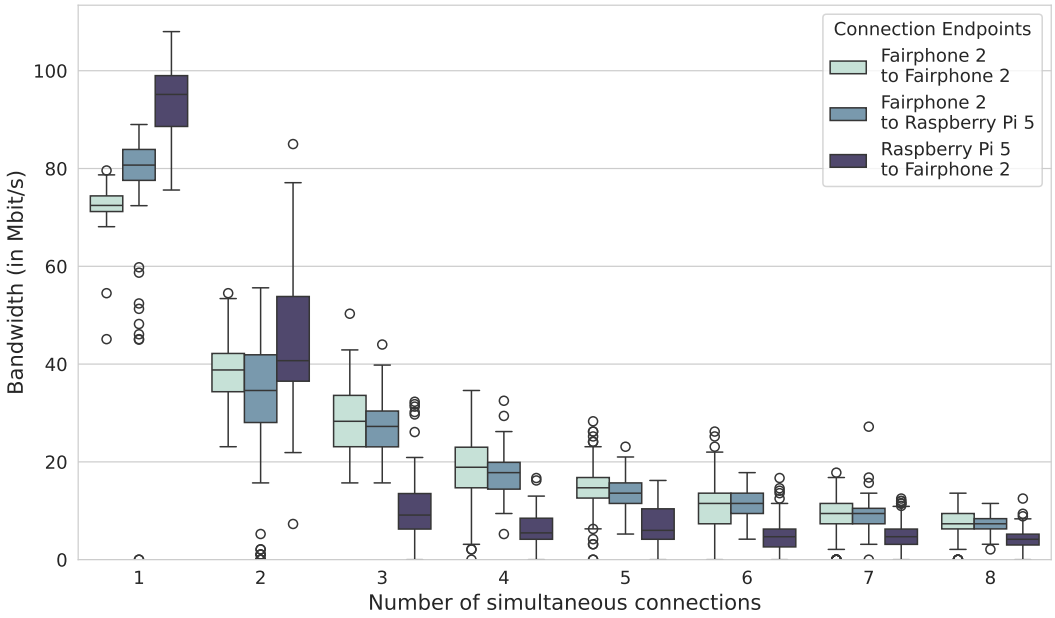

Figure 3.10: Evolution of the goodput per connection between different pairs of active devices for an increasing number of active devices, with a single connection between two active devices, over Wi-Fi 5.0GHz with 20/40/80 channel width using the 802.11ac protocol.

We observe that with the increase of active devices, and thus of connections, the goodput per connection decreases with the following behavior:

$$goodput\_per\_connection < \frac{maximal\_goodput}{2 * number\_of\_connection}$$

where:

- *maximal_goodput* corresponds to the mean between the maximal goodput of the sender and the maximal goodput of the receiver.
- *number_of_connection* corresponds to the total number of corrections, i.e. half of the number of active devices connected to the medium.

The maximal goodput is divided by two since one connection corresponds to 2 devices connected to the medium, one has to send, and the other has to receive. The goodput per connection is less than the computed value because the latter does not take into account things like retransmission or the increasing number of guard intervals.

Each box on the figure represents the aggregation of the measures taken on each connection with the same endpoints. Therefore, the variance and outliers are more than variations of the goodput along the same connection, it also corresponds to unequal division of the total goodput across connections.

For the Ethernet medium, the goodput per connection is much more stable with the increase of active devices, as shown on Figure 3.11.

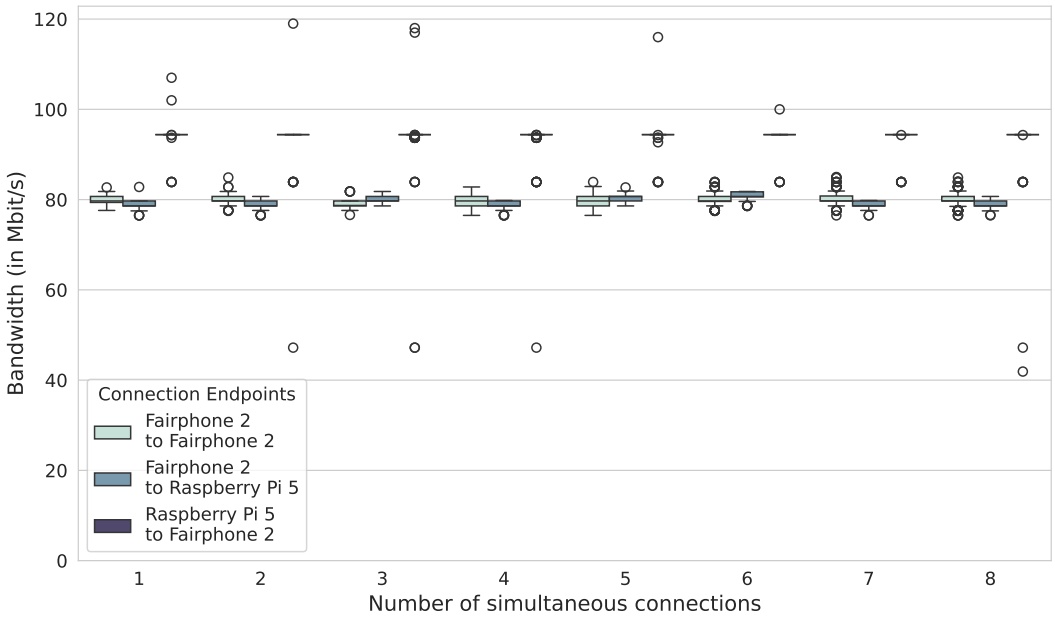

Figure 3.11: Evolution of the goodput per connection between different pairs of active devices for an increasing number of active devices, with a single connection between two active devices, over Ethernet. Micro-USB to Ethernet adapters have been used for Fairphone 2.

Depending on the connection endpoints, the goodput per connection varies between 80 and 95 Mbit/s, and stays the same with the increase in connected active devices. This is explained by the fact that for Ethernet, unlike for Wi-Fi, all devices can transmit at the same time. The only limitations are the maximal throughput of the Ethernet cable, around 40 Gbit/s, the maximal throughput of the switch's ports, around 1 Gbit/s, and the maximal throughput of the router's port, around 1 Gbit/s, and its processing speed. In the worst case in our experiment, the cumulated goodput would be 760 Mbit/s ($= 95 \; Mbit/s * 8$), far from the 1 Gbit/s. Finally, the goodput is equally distributed between

the connections. We still observe some outliers which can be considered measurement errors.

Latency

We will first analyze the latency results without workload, i.e. with *idle devices*, then the ones with the workload, i.e. with *active devices*.

Figure 3.12 show the latency per connection over Wi-Fi 2.4GHz.

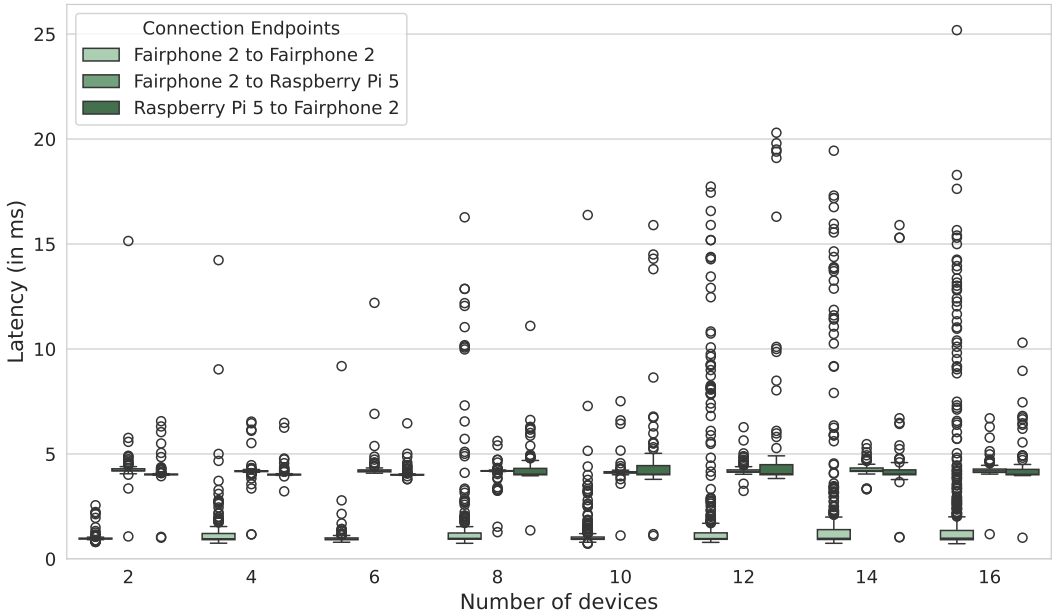

Figure 3.12: Evolution of the latency per connection between different pairs of idle devices for an increasing number of idle devices over Wi-Fi 2.4GHz with 20/40 channel width using the 802.11n protocol.

While the latency per connection stays globally constant with the number of idle devices, we still observe an increasing number of outliers that are always more distant from the median value. This is the result of a higher level of contention between the devices, increasing the latency. Moreover, Wi-Fi 2.4GHz is known to have a low number of frequencies, from 2401 to 2484 MHz, divided into 14 channels of 20MHz width spaced every 5MHz. There is thus overlapping between channels. As the Wi-Fi 2.4GHz frequencies are overcrowded, it is globally accepted that using channels 1, 6 and 11, that do not overlap, leads to less interference between different Wi-Fi. However, in our case, to increase the throughput we have bonded together two channels, leading to more possibilities of interference with other devices and Wi-Fi.

The latency results for Wi-Fi 5.0GHz are similar, as shown on Figure 3.13.

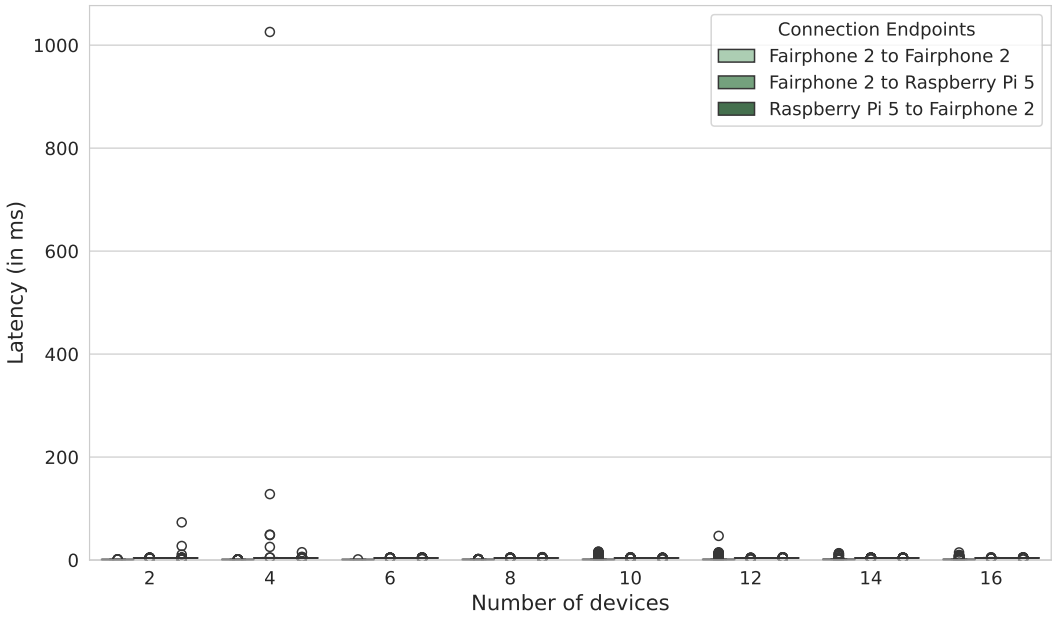

Figure 3.13: Evolution of the latency per connection between different pairs of idle devices for an increasing number of idle devices over Wi-Fi 5.0GHz with 20/40/80 channel width using the 802.11ac protocol.

In contrast with Wi-Fi 2.4GHz, outliers appear later, when 10 idle devices or more are connected. We also observe some extreme outliers for connections between Fairphone 2 and Raspberry Pi.

Without surprise, Figure 3.14 shows that for the Ethernet medium, the latency is way more stable and with fewer outliers. It also stays constant with the number of idle devices connected.

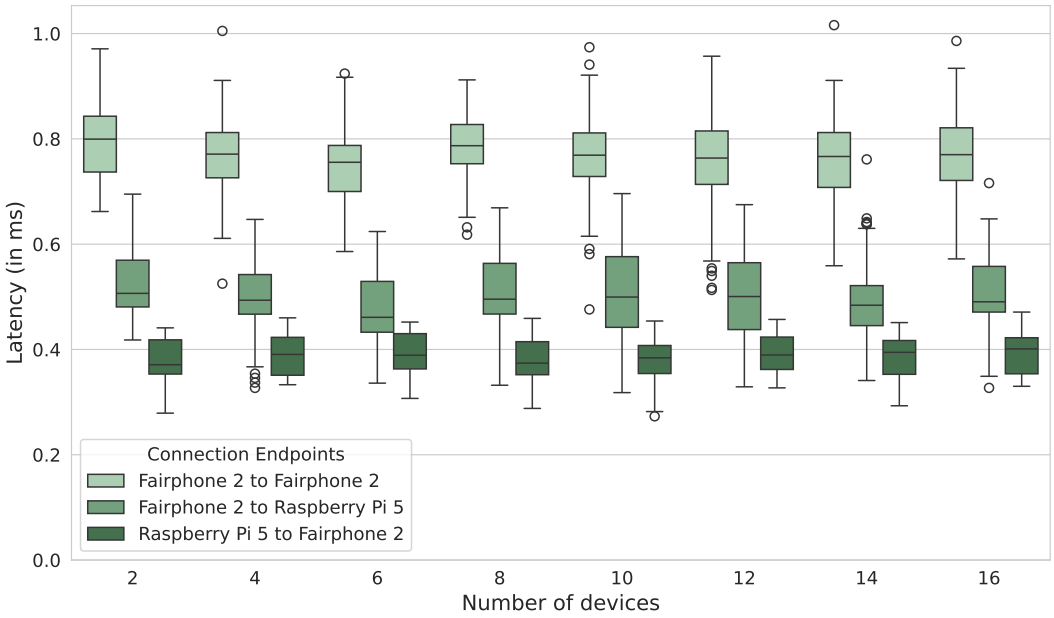

Figure 3.14: Evolution of the latency per connection between different pairs of idle devices for an increasing number of idle devices over Ethernet. Micro-USB to Ethernet adapters have been used for Fairphone 2.

When adding workload between devices, we globally observe an important increase in latency. This is an expected result since the contention and retransmissions will increase when devices become active, i.e. transmit data.

Figure 3.15 and Figure 3.16 show that this increase in latency is even more visible for wireless mediums. For both of them, the latency has passed from 2 5 milliseconds to around 250 milliseconds or more. The global behavior shows that with the increase in connections, the latency increases even more and that more outliers appear. We can notice that most extreme outliers are present on connections between Fairphone 2, showing that they are less robust in environments with higher traffic and interferences. Moreover, Wi-Fi 5.0 counts more and higher outliers than Wi-Fi 2.4, which was not expected as Wi-Fi 5.0 should be more stable. We explain this behavior by the fact that the Fairphone 2 chipset, or its firmware, may not be as performant as the recent chipset when more interference is present. Furthermore, some strange behavior was observed on the Fairphone 2 when connected to Wi-Fi 5.0, such as disconnections or very slow connection of less than one Mbit/s.

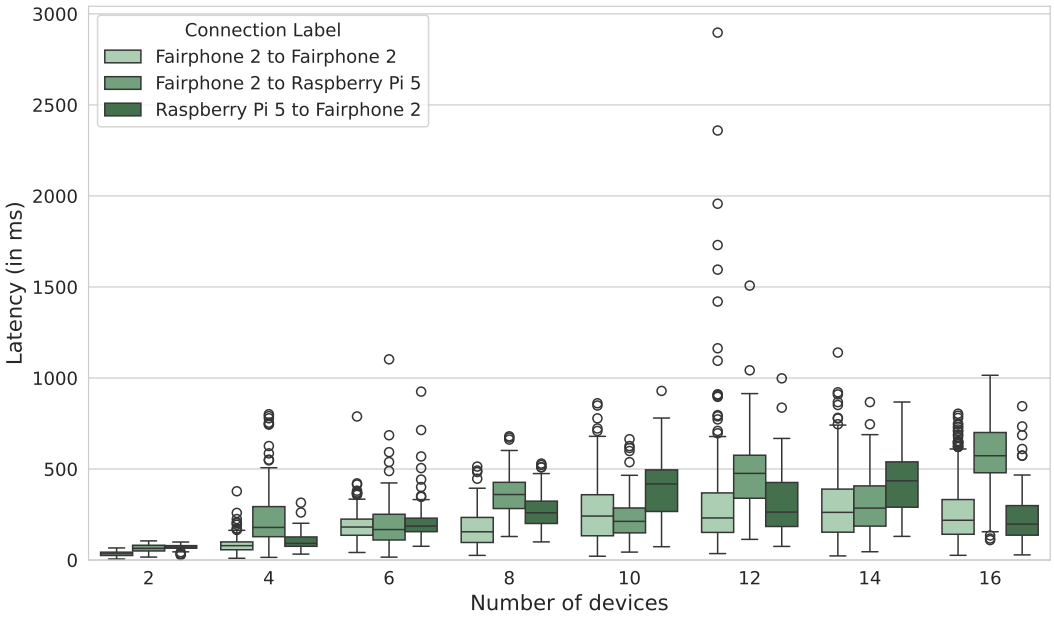

Figure 3.15: Evolution of the latency per connection between different pairs of active devices for an increasing number of active devices over Wi-Fi 2.4GHz with 20/40 channel width using the 802.11n protocol.

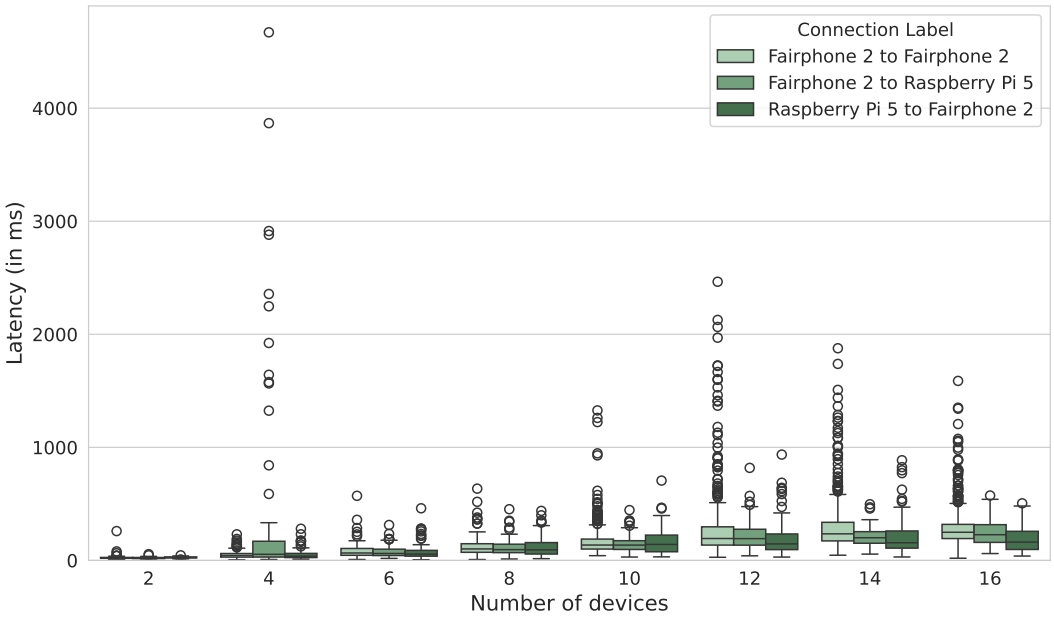

Figure 3.16: Evolution of the latency per connection between different pairs of active devices for an increasing number of active devices over Wi-Fi 5.0GHz with 20/40/80 channel width using the 802.11ac protocol.

Finally, for the Ethernet medium with the workload, Figure 3.17 shows a less important increase in latency than for wireless mediums.

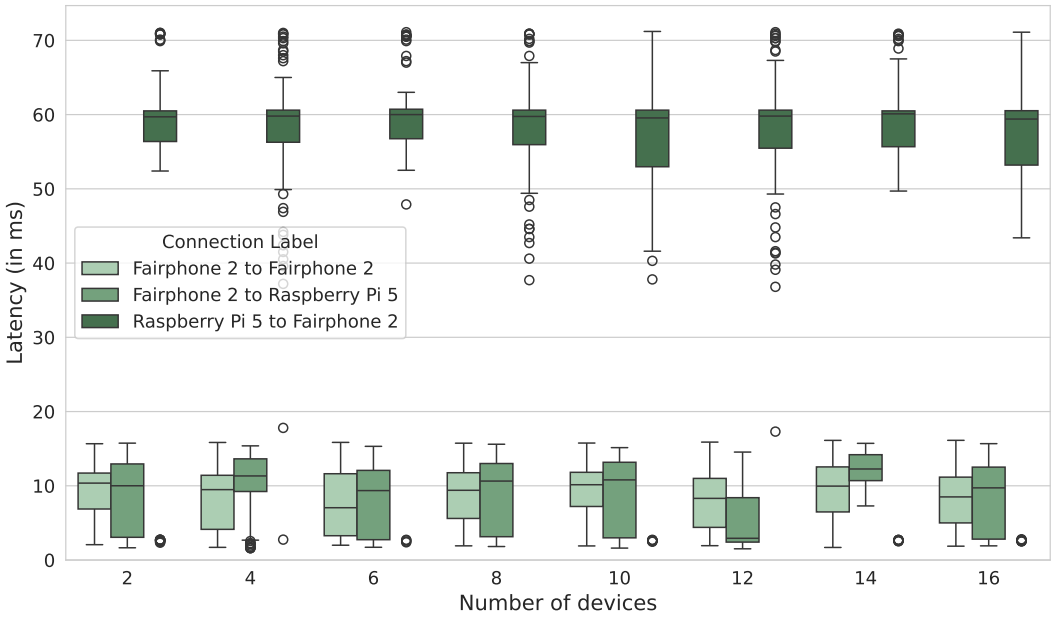

Figure 3.17: Evolution of the latency per connection between different pairs of active devices for an increasing number of active devices over Ethernet. Micro-USB to Ethernet adapters have been used for Fairphone 2.

We can also notice a clear increase in latency when the Raspberry Pi 5 is the sender. This is explained by the fact that the Raspberry Pi 5 sends data way faster than the Fairphone 2 is capable of receiving, leading to more contention and an increase in latency.

Conclusion

This experiment demonstrates that the network performance of the devices is way more sensible to the increase of connected devices, whether active or inactive, over wireless mediums than over wired mediums. The goodput per connection becomes extremely reduced and the latency becomes higher, less stable and shows large peaks occurring at random.

In comparison, the wired medium didn't show any loss of goodput and only a moderate increase in latency, compared to wireless mediums, which stay stable. However, these results were only possible because the hardware limits of the devices composing the experimental setup were not reached during the experiment.

3.5 Conclusion

This preliminary network evaluation shows that old smartphones, such as the Fairphone 2, have enough networking capabilities to be reused for IoT applications or cluster applications. They can compete with other recent embedded devices, such as the Raspberry Pi 5, over wireless mediums such as Wi-Fi 2.4GHz and Wi-Fi 5.0GHz. However, they are not as performant as them over Ethernet, mainly due to the use of outdated technologies like USB 2.0 which has been replaced by USB 3.0, offering better performance. By using the correct adapters, it may still be a viable alternative for applications that do not require significant data transmission.

Increasing the number of devices and connections results in an important diminution of the per-connection performance over wireless mediums; the goodput is significantly reduced while the latency greatly increases, becomes less stable and shows random peaks. Furthermore, disconnections and slow connections of less than one Mbit/s were randomly observed, in particular for Wi-Fi 5.0GHz. The latter may have been resolved with more recent technologies. Over the Ethernet medium, devices have kept the same goodput, but their latency has increased, moderately in comparison with wireless mediums. However, the Ethernet results must be treated with caution, as the hardware limits of the equipment have not been reached. A similar behavior as the one for wireless medium could have been observed if it had been the case.

Finally, we've observed differences in goodput and latency depending on the connection endpoints and the medium used. These are the results of differences in goodput and latency between the Fairphone 2 and the Raspberry Pi 5 depending on the networking mediums. This could have an impact on the cluster, depending on its architecture.

Chapter 4

The cluster

This chapter is about the cluster design and architecture. We've designed and built two variations of the cluster, one using only Fairphone 2 while the other uses a restrained number of Raspberry Pi 5 for key roles in the cluster. The chapter is structured as follows. First, we start by talking about *K3S*, its requirements and the limitations encountered. We then give a preliminary analysis of the impact of running it on the devices. Finally, we describe the two variations of the cluster, their topology and how they are configured.

4.1 Devices setup

Before creating the cluster, the first step is to set up and configure the devices that will be part of it. Two kinds of devices will be used, the Fairphone 2 and the Raspberry Pi 5.

PostmarketOS offers a community port under testing adapted for Fairphone 2 but only allowing partial functionalities of the device. Therefore the smartphones operate under a `postmarketOS edge` image with the Linux kernel version `6.9.1`. Furthermore, we had to recompile the given kernel to add the `Multiple port match support` kernel module, the reasons for this are discussed in the next section. For Raspberry Pi 5, we opt for the `Debian GNU/Linux 12 (bookworm)` with the Linux kernel version `6.6.31-v8-16k+`.

For the networking part, we have deactivated the firewalls on each device to avoid messing with the K3S framework. As we kept the project in a development and testing phase, this simplification could be done. For a deployment, the firewall should be activated and adapted.

Furthermore, as discussed in chapter 3, we ensure that the *powersave mode* for the Wi-Fi was disabled, ensuring better Wi-Fi performance.

To power the smartphone, we removed the battery and connected the pin of it to a USB cable. The phones being powered through the battery pin allows the micro-USB port to stay available for an Ethernet connection. This avoids the need for a USB-Ethernet adapter with powering.

4.2 K3S on devices

In this work, we use the K3S to build the cluster of smartphones. While being a lightweight version of Kubernetes, the framework still has some kernel and hardware requirements that should be met. In this section, we will discuss such requirements and how they apply

to our devices. We will also analyze the impact it has on each device, depending on its role in the cluster, i.e. server or agent.

Kernel requirements

K3S depends on several optional and mandatory kernel modules to run correctly. The `Multiple port match support` [41] is a mandatory kernel module to K3S, required for *Traefik* and *ServiceLB* to run. This module allows writing rules where TCP or UDP packets are matched based on a series of source or destination ports instead of a single one. However, this kernel module was not part of the Linux kernel provided in the postmarketOS port of the Fairphone 2.

Fortunately, postmarketOS allows us to easily change the kernel configuration to add a missing module and rebuild the kernel. The missing module was therefore added to the smartphone.

This issue highlights that compatibility is not ensured between the OS and frameworks, indicating that modifications and adaptations may be necessary to meet our needs.

Hardware requirements

The hardware requirements to run K3S are not too restrictive. For a small cluster of some nodes, it is recommended to have at least a two-core CPU and 1 GB RAM. However, for bigger or high-availability clusters, the minimum CPU and memory requirements increase as shown in Table 4.1.

Deployment Size	**Nodes**	**VCPUS**	**RAM**
Small	Up to 10	2	4 GB
Medium	Up to 100	4	8 GB
Large	Up to 250	8	16 GB
X-Large	Up to 500	16	32 GB
XX-Large	500+	32	64 GB

Table 4.1: K3S minimal CPU and memory requirements for server nodes depending on the size of the cluster [42].

If a Fairphone 2, which only has 2 GB of memory, is used as the server node, the cluster should not be able to reach a size of 10 nodes. This is an important theoretical limitation of K3S in our case, as we are limited to the Fairphone 2 phone model in this work. To solve this limitation, we use Raspberry Pi 5, with 8 GB of memory, to replace Fairphone 2 in the role of the server node. Doing so should allow us to reach a cluster of medium size, with around 100 nodes. Even if not an upcycled device, the Raspberry Pi 5 can be considered a substitute for more recent upcycled smartphones with higher specifications.

For agent nodes, Fairphones 2 are used in both variants.

4.3 Cluster design

Let's now deep into the cluster design. We've designed two variants of the cluster, one is using a Fairphone 2 as server node, while the other is using a Raspberry Pi 5. Both

variants only use Fairphone 2 as agent nodes. In this section, we will give the cluster architecture, explaining how the nodes are connected. We then give the details on the K3S configuration used to deploy our application on the cluster.

4.3.1 Architecture

Due to the low computing power of the Fairphone 2, we assume that a cluster with this smartphone model wouldn't be adapted for applications with too high requirements. Therefore, we opt for a simple cluster architecture, without multiple server nodes that would ensure high availability, nor internal or external databases.

Unlike *Kubernetes*, *K3S* comes with a set of default services that can or not be replaced by others if desired. In this work, we kept all default services, that are:

- *Flannel* as the container network interface (CNI).
- *Traefik* as the Ingress controller and for routing the HTTP/HTTPS traffic to the appropriate service with the cluster.
- *ServiceLB* as the load balancer for `LoadBalancer` service.
- *Containerd* as the container manager.

As we do not implement a high-availability cluster, a single server node is needed. Agents node, connected to the same networking medium, join the cluster through the server. The routing between devices is managed by an external router while routing inside the cluster is managed by the server node (and then by the external router).

The Figure 4.1 represents the architecture used for this cluster, without any device needed for network communications such as the router or the switch.

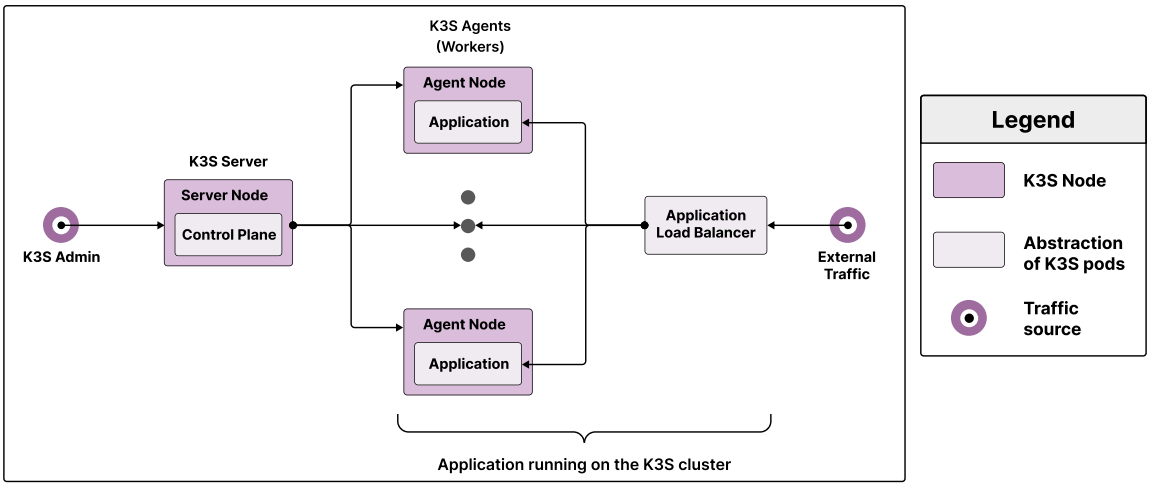

Figure 4.1: Architecture of the cluster. Abstraction of K3S pods corresponds to pods deployed for the corresponding purpose. The *Application Load Balancer* is not part of any nodes for a representative purpose. In reality, this service has one pod running inside each node that is part of the linked deployment. This architecture was inspired by one of the architecture setups proposed by *K3S* [43].

External traffic destined for the application is handled by *ServiceLB*. Each node that is part of the application deployment runs a pod of the load balancer service. These pods

are responsible for updating the internal routing table of the node to forward requests to other nodes of the deployment. It also exposes the IP address of the nodes that are part of the deployment, the one of server nodes and any additional provided IP. As a result, any node that is part load balancer service, even if they can not run deployment pods, can receive incoming external traffic and forward it to other nodes.

We did not change this principle as it is the default behavior of *ServiceLB*. However, for the rest of this document, we will consider the server node IP as the only way inside the cluster. It can be considered as the gateway of the cluster by which all the external traffic passes.

As *Flannel*, *Traefik* and *Containerd* are managed implicitly by K3S and as their configuration is not part of the design, we won't detail them more than what was explained in the background section (subsection 2.2.4).

4.3.2 K3S Configuration

We will now discuss the *K3S* configuration used for the deployment of the application and its load-balancing service. This step takes place once the cluster is formed; that is when agent nodes have joined the cluster through the server node.

Let's first take a look at the `YAML` configuration file of the application deployment on listing 2.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
name: k3s-tflite-app-deployment
spec:
replicas: 1
selector:
    matchLabels:
    app: k3s-tflite-app-pod
template:
    metadata:
    labels:
        app: k3s-tflite-app-pod
    spec:
    containers:
    - name: k3s-tflite-app-container
        image: k3s-tflite-app-image
        imagePullPolicy: Never
        ports:
        - containerPort: 18080
    affinity:
        nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
            - key: disallow-deployment
                operator: NotIn
                values:
                - "true"
```

Listing 1: `YAML` configuration file of the application deployment.

It corresponds to a *Kubernetes Deployment* named `k3s-tflite-app-deployment` with a single replica by default. It runs pods identified as part of the deployment, i.e. labeled `k3s-tflite-app-pod`.

This pod is configured to run a container, named `k3s-tflite-app-container`, running the image `k3s-tflite-app-image` and exposing the port 18080, which corresponds to the port used by the server of our application. Notice on line 18 that the image is pulled from anywhere, instead a local image should be imported to the agent nodes, from a `tar` archive for example. This was done for convenience but can be changed if desired. The `Dockerfile` from which the image was built can be found in Appendix A.

Lastly, an affinity was added to the pods, such that they would only be scheduled and deployed on nodes that do not have the label `disallow-deployment` to true. This was done to prevent application pods to be running on the server node, as per our design, by labeling it with `disallow-deployment=true`.

A service was also configured to work with the deployment and allow load balancing of the HTTP/HTTPS requests between the nodes/pods. Listing 2 shows its `YAML` configuration

file.

```
apiVersion: v1
kind: Service
metadata:
name: k3s-tflite-app-service
spec:
type: LoadBalancer
selector:
    app: k3s-tflite-app-pod
ports:
- protocol: "TCP"
    port: 18080
    targetPort: 18080
```

Listing 2: `YAML` configuration file of the load balancing service for the application deployment.

It corresponds to a load balancing service named `k3s-tflite-app-service`. It will expose the application externally, listening on port 18080 for TCP connections to redirect the external traffic to port 18080 inside the pods labeled `k3s-tflit-app-pod`. Furthermore, it will distribute the external traffic between nodes running those pods.

Chapter 5

TensorFlow Lite application

This chapter chapter is about the TensorFlow Lite application that will be running on the cluster and used as a validation of its capabilities. We start by discussing some compatibility issues that were encountered when compiling TensorFlow Lite for the phones under postmarketOS. We then describe the application and how we have adapted it for our needs, especially by adding networking capabilities to it using the *Crow* framework.

5.1 Compatibility with *musl libc*

TensorFlow Lite, the lightweight version of TensorFlow, is designed to only perform inference on edge devices. TensorFlow Lite does not have a pre-compiled package available for each device. Instead, they provide a compilation toolchain to compile the framework to your own iOS, Android or ARM device. Like TensorFlow, TensorFlow Lite is implemented based on the *glibc*, an implementation of the C standard library, which is different than *musl libc*, another implementation used by postmarketOS. *Musl libc* follows strictly the C standards, while *glibc* allows itself to add some new function on top of the standard for convenience. These additional functions result in compatibility issues when running software and frameworks implemented with *glibc* on *musl libc* platform, as some of these additional functions are not recognized.

As the Fairphone 2 is running with postmarketOS, based on *musl libc*, we add to resolve this issue. To do so, we lightly modify the TensorFlow and TensorFlow Lite codes to remove or change parts that used unknown functions to *musl libc*. We did not test if the entire framework was compatible with *musl libc* but restricted ourselves to the parts used by our application. We used the release `2.16.1` of TensorFlow, for which we had to adapt around 50 lines of code across the project. The difficulty of such an adaptation does not reside in the number of lines that have to be changed, but rather in determining which lines should be modified and how they should. As modifications are version-dependent, they will not be described here but instructions can be found on the GitHub repository of this project. After cross-compilation of TensorFlow Lite with the flags adapted for the Fairphone 2, we obtain a `C/C++` static library. It can be added, with the corresponding headers, to any `C/C++` projects that require TensorFlow Lite.

5.2 Label image application

More than just showing the capabilities of a smartphone cluster, this work also aims to demonstrate the use of frameworks and software, particularly TensorFlow Lite, for edge applications on devices that are not especially suited for such tasks. Our application is based on the already implemented image classification example provided by TensorFlow Lite [44]. It consists of inferring the most plausible objects present on a bitmap image, in the limit of the labels that can be recognized by the model.

The provided implementation is a monolith design for a single complete run. The main function first sets up the interpreter by loading the `tflite` model used for inference, then configures and builds the interpreter, the profiler and the delegations. After the setup phase, the running phase starts by reading and resizing the bitmap image to be compatible with the model. The interpreter can be warmed up, which is optional, before performing the inference on the image. Finally, results are retrieved and processed to match labels before being displayed.

We've adapted the given implementation to make it runnable on the cluster. On the one hand, we've made the application more modular by dividing the main function into two functions, one to set up everything, and the other to perform the inference. Doing that allows the interpreter, the profiler and the delegations to be built only once at the start, and not before each inference. On the other, we've added networking capability by including a web server using the lightweight framework *Crow*. As it is also written in `C++`, it can easily be integrated into our code. Adding this allows the application to handle HTTP POST requests containing the bitmap image on which the inference should be performed. Since Crow lacks the functionality to handle requests containing images, we had to implement it ourselves. The implementation of this function can be found in Appendix B.

In the final version of our application, we added a web server phase between the setup phase and the running phase. The flow is as follows. The application sets up everything a single time at the start, by configuring the interpreter, profiler, delegations and crow webserver from the given parameters. The `tflite` model used for inference, as well as the text file containing the corresponding labels should be available locally on the device. Once the setup phase is over, the inference part works by receiving HTTP POST requests containing an image that should be labeled. The request is handled by the Crow web which temporarily stores it locally during the inference. The inference is then performed on the image and the resulting labels, as well as their respective confidence score, are returned in response to the HTTP request.

The application takes as argument a `JSON` file containing the configuration settings for the TensorFlow Lite part and for the Crow server part. We disable most of the parameters and only consider the number of threads used for inference and the number of threads used for the Crow web server. Furthermore, we only perform inference using the CPU of the Fairphone 2, as the postmarketOS port only renders the GPU partially functional and in any case, GPU acceleration is not supported by TensorFlow Lite for our device. We also disable the profiler and the warm-up of the interpreter and reduce the number of inference loops to a single one. Our application has been designed to validate the feasibility of a cluster of smartphones using *K3S*. It also demonstrates the possibility of running TensorFlow (Lite) on upcycled smartphones under Linux. Therefore, it was not designed nor parametrized to be as performant as possible.

Chapter 6

Evaluation

In this chapter, we evaluate the performance of the cluster using the TensorFlow Lite application. The chapter is divided into 3 sections. First, we define the testbed with the common parametrization used in all the experiments. We then evaluate the performance of the TensorFlow Lite image classification application as the cluster depends on it. Afterward, we evaluate the performance of both cluster variants, for different networking mediums and cluster size. Finally, we conclude with a discussion about the results of the evaluation.

6.1 Testbed

As some of the following experiments are similar, we will define, in this section, parts, default settings as well as performance metrics common to several experiments.

Starting with the script and tools, the two first experiments of the evaluation of the TensorFlow Lite Application measure the response time of the application, in microseconds. In these experiments, the client, i.e. the *HP Probook G6 450*, sends a `HTTP POST` request containing a bitmap image to the *Crow* web server of the TensorFlow Lite application using `curl`. The measurement of the response time is done with the `time` Linux function, which measures the execution time of a command. To measure the execution time of some of the application parts or subparts, the `execution_duration` flag can be set to add execution information to the result of the requests.

The last experiment of the TensorFlow Lite application evaluation, as well as the experiment of the cluster evaluation, measures the number of requests that are handled per second. This time the client, i.e. the *HP Probook G6 450*, simulates a workload of HTTP requests using the modern HTTP benchmarking tool `wrk` [45].

We configure `wrk` to be run with 8 threads, as the laptop is limited to 8. The workload lasts for 20 seconds during which up to $10 * \#agents\_nodes$. As you can observe, the number of connections is dynamic and depends on the number of agent nodes currently in the cluster. We made this choice because having a fixed number of connections does not represent correctly the scaling of the cluster. If the number of connections is too high, the Crow web server is saturated, leading to bad performance. On the other hand, if the number of connections is too low, some threads of the application won't have any work, also leading to bad performance. The dynamic value of connection is based on the number of connections for which a single agent running the application performs the best, multiplied by the number of agents in the cluster. Therefore, it should represent the best number of connections for the current cluster size. If the number of requests per second

decreases, it means that the cluster did not scale properly.

Finally, we've added a `lua` script that allows `wrk` to create `HTTP POST` request that includes a bitmap image. This script is given in listing 3.

```lua
-- Post request inspired of:
↪ https://gist.github.com/tonytonyjan/d2a612f2b3f37837fc4d5c1409ac0b1e

-- Load image data from file
local filename = "grace_hopper.bmp"
local file_path = "./assets/grace_hopper.bmp" -- relative to the
↪ experiment notebook
local file = io.open(file_path, "rb")
local file_data = file:read("*all")
file:close()

-- Generate the multipart body
local boundary = "-------YouShallNotPassBondary";
local part_name = "image"
local content_type = "application/octet-stream"
local crlf = "\r\n"

local content_disposition = 'Content-Disposition: form-data; name="'
↪ .. part_name .. '"; filename="' .. filename .. '"'
local content_type = 'Content-Type: ' .. content_type

-- Create the HTTP request
wrk.method = "POST"
wrk.body = "--" .. boundary .. crlf .. content_disposition .. crlf ..
↪ content_type .. crlf .. crlf .. file_data .. crlf .. "--" ..
↪ boundary .. "--" .. crlf
wrk.headers["Content-Type"] = "multipart/form-data; boundary=" ..
↪ boundary
```

Listing 3: `lua` script to generate an HTTP POST request containing a bitmap image.

For all experiments, the same bitmap image was used, a picture of Grace Hopper [46] of dimension 517x606 with a color depth of 24 bits/pixel, for a total size of 940,650 bytes. All experiments have been run over the three same networking mediums as in chapter 3:

- **Wi-Fi 2.4GHz**: The wireless medium Wi-Fi 2.4GHz with 20/40MHz channel width, only authorizing the Wi-Fi protocol 802.11n (Wi-Fi 4).
- **Wi-Fi 5.0GHz**: The wireless medium Wi-Fi 5.0GHz with 20/40/80MHz channel width, only authorizing the Wi-Fi protocol 802.11ac (Wi-Fi 5).
- **Ethernet**: The wired medium Ethernet. Note that adapters are used to adapt the micro-USB port of the smartphones to the Ethernet port of the switch.

All experiments in the TensorFlow Lite application evaluation have been performed 20 times for each networking medium. The application was restarted after each run to ensure that no requests from the previous run remained.

The experiment of the cluster evaluation is performed two times, once per cluster variant, overall the three networking mediums. For each networking medium, we performed 10 runs per cluster size, from 2 to 9, resulting in a total of 80 runs per medium. Between each run, all application pods are restarted and the next run only started when all previous pods were deleted and new ones are running. As for the previous experiment, this was done to ensure that no previous requests remained. Furthermore, when not part of the cluster, all devices are shut down to reduce the risk of interference.

Each agent nodes only run a single application pod, as it uses mostly all resources of the device.

6.2 TensorFlow Lite Application

The evaluation of the TensorFlow Lite application consists of a total of three experiments. These experiments have for purpose of characterizing the application, by determining the execution time of each time. They also show the evolution of performance when we vary the two selected parameters, i.e. the number of threads used for the inference and the number of threads used by the *Crow* web server. Globally, they provide a basis for benchmarking the cluster.

6.2.1 Experiment 1: Execution time of each application part and subpart

In this experiment, we analyze the execution time of each part of the modified image classification application with networking capability, described in section 5.2. The experiment consists of sending HTTP requests for inference of an image to a single Fairphone 2 running the TensorFlow Lite image classification application directly, i.e. not in a container or a cluster. The experiment has been performed over each networking medium defined in the testbed section (section 6.1). For each medium, 20 iterations were performed instead of the 10 default ones. An iteration consists of starting the application through SSH and sending an HTTP request to it using `curl`. Execution times are measured by the application itself and added to the results when started with the `execution_duration` flag set to 1.

As the variation of medium is of poor interest for this experiment, we will only show and discuss the results for the *Ethernet* medium here. Anyway, the experiment was done for the three mediums and shows similar results that can be found in Appendix C.

Figure 6.1 shows in detail the execution time, in microseconds, for each subpart of the three main parts of the application. Even if five subparts particularly stand out, the **interpreter ivnvocation** takes the most time. It corresponds to the time taken by the interpreter to perform all inference loops, a single one in our case. Other time-consuming subparts are the three ones working with the bitmap image, **handle POST request with image**, **read imagefrom file** from the file and **resize and copy image to the input tensor**. Finally, the **configuration and build interpreter** takes the less amount of time between the 5 discussed subparts.

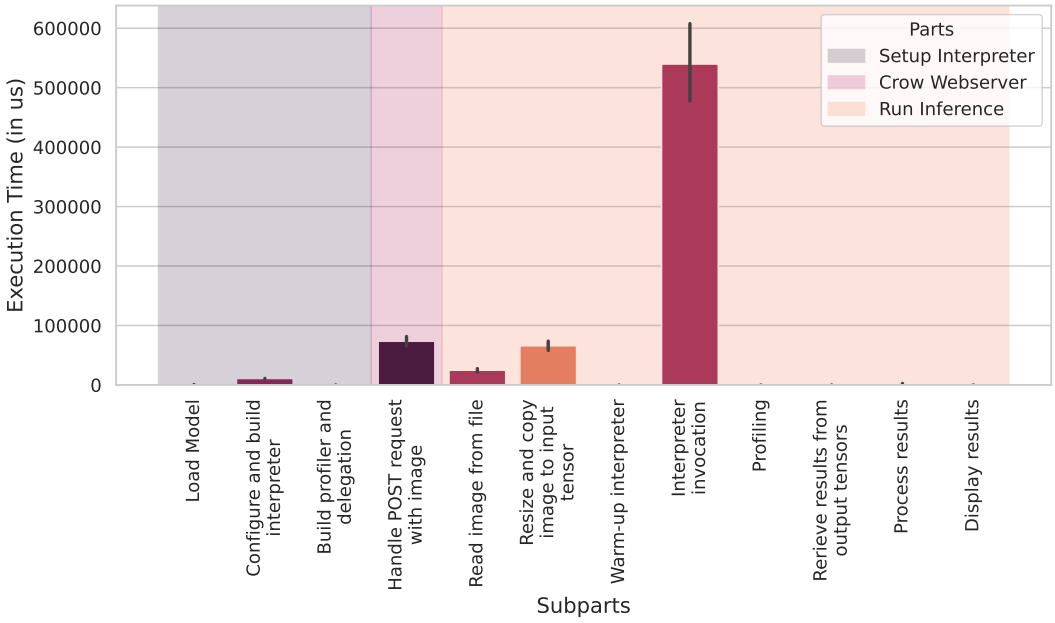

Figure 6.1: Comparison between the execution time in microseconds of each subpart of the TensorFlow Lite image classification application over the Ethernet medium. Subparts have been grouped according to the part of the application to which they belong and have been sorted in execution order.

A summarized view of the execution time, in microseconds, per part and compares them to the total execution time is given in Figure 6.2, while Figure 6.3 shows the ratio of the total execution time of each part.

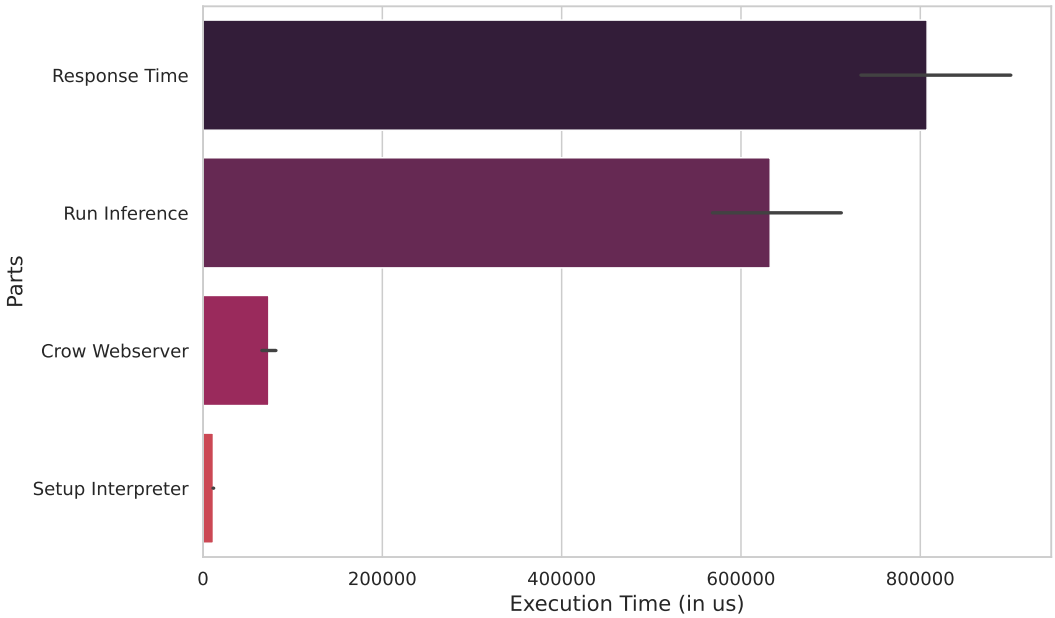

Figure 6.2: Comparison between the execution time in microseconds of each part of the TensorFlow Lite image classification application and the total execution time, over the Ethernet medium.

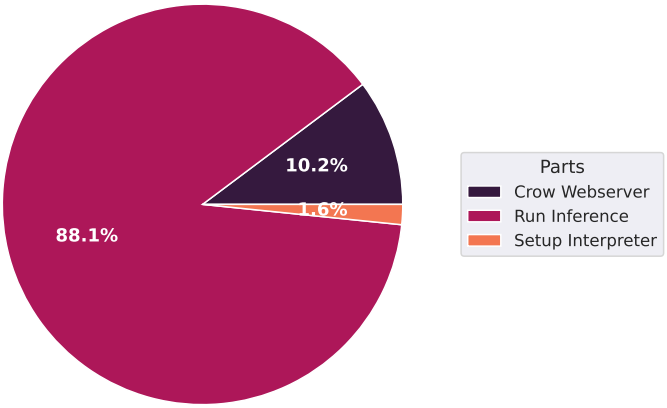

Figure 6.3: Ratio of the total execution time for each part of the TensorFlow Lite image classification application over Ethernet

We can observe that while setting up the interpreter only takes around 1.6% of the total execution time of a request, for an important number of requests it may still result in an important loss of time.

6.2.2 Experience 2: Evolution of response time with different number of inference threads

The number of threads used by the image classification application is one of the two parameters that we've kept. In this experiment, we analyze, over each medium, how the response time evolves for an increasing number of threads.

The experiment measures in microseconds the response time of a single HTTP request. Figure 6.4 shows a comparison of the response time, per medium and the number of inference threads. We observe that the lowest response time is for a single inference thread. For a greater number of inference threads, the response time is higher. This may sound counter-intuitive as we would expect the inference time to go faster with more threads. However, in our case, we limited the number of inference loops to one, meaning that instead of parallelizing the loop executions between a greater number of inference threads, we lose time with useless threads.

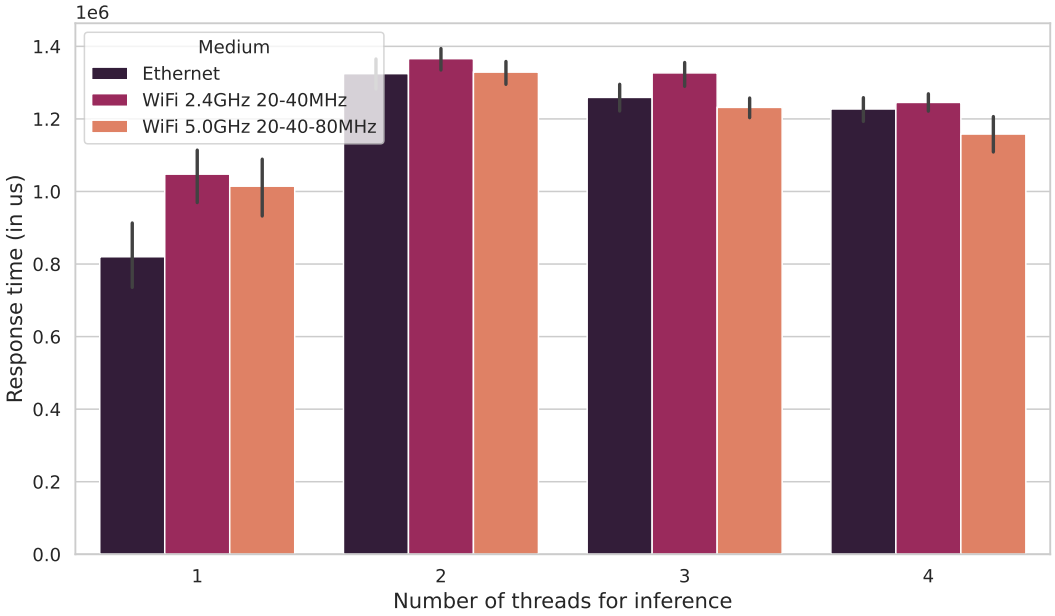

Figure 6.4: Evolution of the response time in microseconds for an increasing number of inference threads, over different networking mediums.

6.2.3 Experiment 3: Evolution of the number of requests per second for different numbers of server threads

Our application is designed to be a web server that handles requests by performing inference on the images that they contain. In this experiment, we will analyze the impact of the number of threads allocated to the server on the number of requests per second that it can handle. As by default the Crow server is already two-threaded, with an acceptor and a worker thread, we will make the number of threads vary from two to four. This increases the number of worker threads from one to three, and the same should happen from the number of requests that can be handled in parallel.

The results of the experiment can be seen on Figure 6.5. We observe a linear increase in the number of requests handled by second with the increase of the number of server threads. More specifically, the number of handled requests per second is multiplied by the number of worker threads. This is expected since it corresponds to the number of requests treated in parallel. Globally, the networking medium does not have a huge impact on the results.

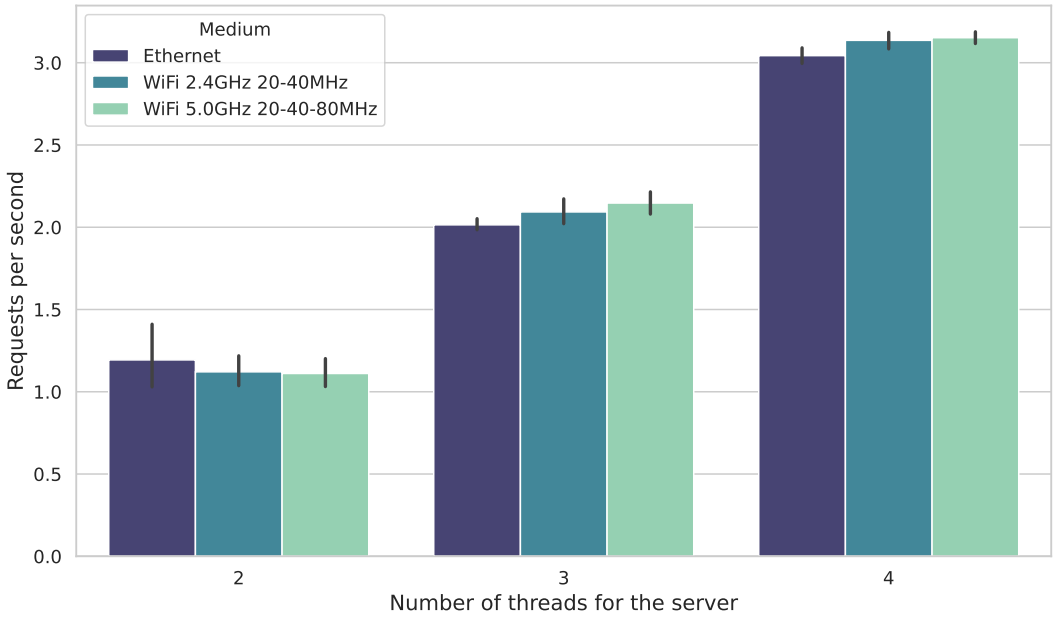

Figure 6.5: Evolution of the number of requests per second for different numbers of server threads, over different networking mediums.

6.3 Cluster

Now that we have evaluated the performance of the TensorFlow Lite application that will run on the cluster, let's evaluate the performance of the cluster itself.

6.3.1 Experiment 1: Evolution of the number of requests per second for different cluster size

This experiment consists of determining the evolution of the number of requests per second depending on the cluster size, for the two cluster variants over the three mediums. The parameters of the TensorFlow Lite application running on the cluster are the ones that have shown the best results in the previous experiments (subsection 6.2.2 and subsection 6.2.3), which are a single inference thread and four server threads. As per the cluster design, we made its size increase by adding new agent nodes. The cluster size varies from two, one server node and one agent node, to nine, one server node and eight agent nodes. We limited our cluster to nine nodes, due to hardware limitation of the Fairphone 2 for the use of *K3S*. We do the same for the Raspberry Pi 5 variant for compliance.

Fairphone 2 variant

The results for the Fairphone 2 cluster variant are shown in Figure 6.6.

For each medium, the number of requests handled per second increases with the number of agent nodes at first, until reaching a medium-specific threshold after which the performance decreases. The Ethernet medium shows the best performance, with a threshold of around 7.5 requests/s for four to five agent nodes. It is also the medium with the sharpest increase and the slower decrease.

On the other side, wireless mediums show more moderate performance. The Wi-Fi 5.0GHz displays relatively good performance in comparison to the Ethernet medium. Its threshold is around 6.5 requests/s for 4 agent nodes. However, it decreases faster than the wired medium once the threshold is reached.

Finally, the cluster over Wi-Fi 2.4GHz shows very poor performance. Its threshold is less than 3 requests/s, and it responds to almost no request when the cluster has 7 and 8 agent nodes.

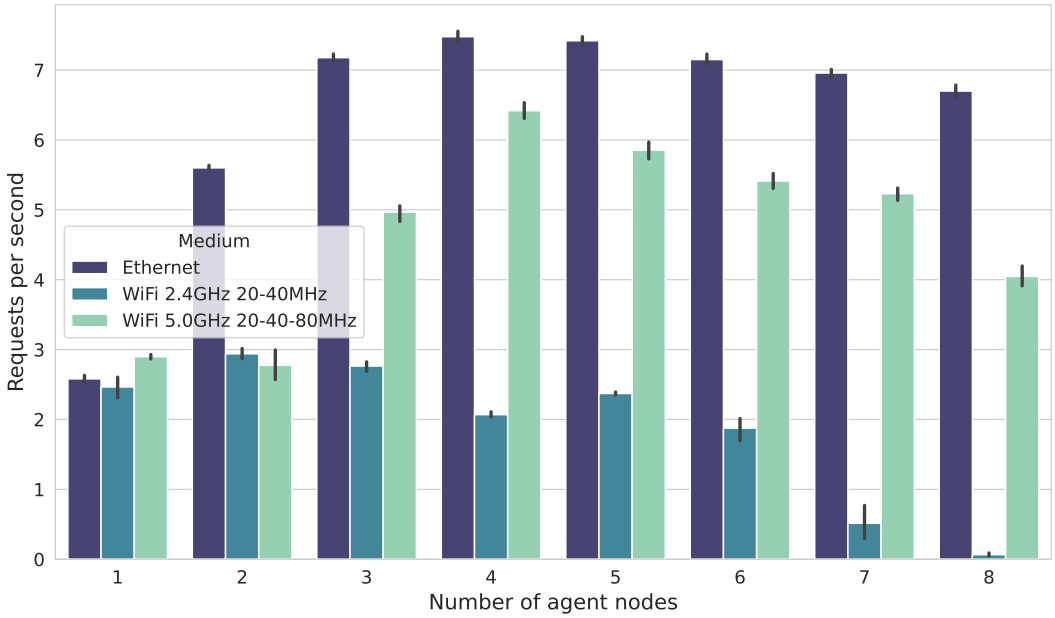

Figure 6.6: Evolution of the number of requests per second for an increasing number of agent nodes in the Fairphone 2 cluster variant over different networking mediums.

The observed results correspond to what was expected from the previous evaluations. The preliminary evaluation of the networking performance of the Fairphone 2 has shown that Ethernet was the best medium, as the goodput stable and the latency stays relatively low when the number of connections and devices increases. In contrast, Wi-Fi mediums had shown an important diminution in goodput per connection and an increase in latency when the number of connections and devices increased.

As the wireless mediums have to take turns to send and receive, the total goodput has to be divided between the connections. The server node receives with greater difficulty the HTTP requests from the router and has more difficulty forwarding them to the agent nodes. Furthermore, as wireless mediums are more vulnerable to interferences, the performance of the cluster over them is more prone to decrease with the cluster size. This is particularly the case for Wi-Fi 2.4GHz which, on top of being overloaded by other devices, uses 40MHz that are more prone to interference due to co-channels.

Raspebrry Pi 5 variant

The results for the Raspberry Pi 5 cluster variant are shown in Figure 6.7. As we look at the behaviors per medium, we observe some significant differences with the Fairphone 2 variant. First, for the Ethernet medium, the number of requests per second only increases with the number of agent nodes, which can be explained by looking at the preliminary

networking results (see Figure 3.4). The Raspberry Pi 5 has a maximal goodput of around 950 Mbit/s, much better than the 90 Mbit/s of the Fairphone 2. Therefore, it can receive way more data but also send way more data. Furthermore, Ethernet is capable of sending and receiving at the same time, meaning that the Raspberry Pi 5 can send at 950 Mbit/s while receiving at the same speed. In contrast, the Fairphone 2 is quickly at its limit, because of the lower goodput but also because it only has a single data transfer channel, meaning that it must alternate between sending and receiving.

For the wireless mediums, Wi-Fi 5.0GHz has a similar behavior similarly to the Fairphone 2 cluster variant, but with a lower number of requests per second. This is explained by the lower goodput from a Raspberry Pi 5 to a Fairphone 2 in comparison to the goodput between Fairphone 2, as observed on Figure 3.10.

Finally, the Raspberry Pi 5 cluster variant over Wi-Fi 2.4GHz shows rather disappointing performance, worse than those of the Fairphone 2 variant. We assume that it is for the same reason that for the other cluster variant, combined with the lack of support for 40MHz channel width over this medium, leading to lower performance as seen in Figure 3.2.

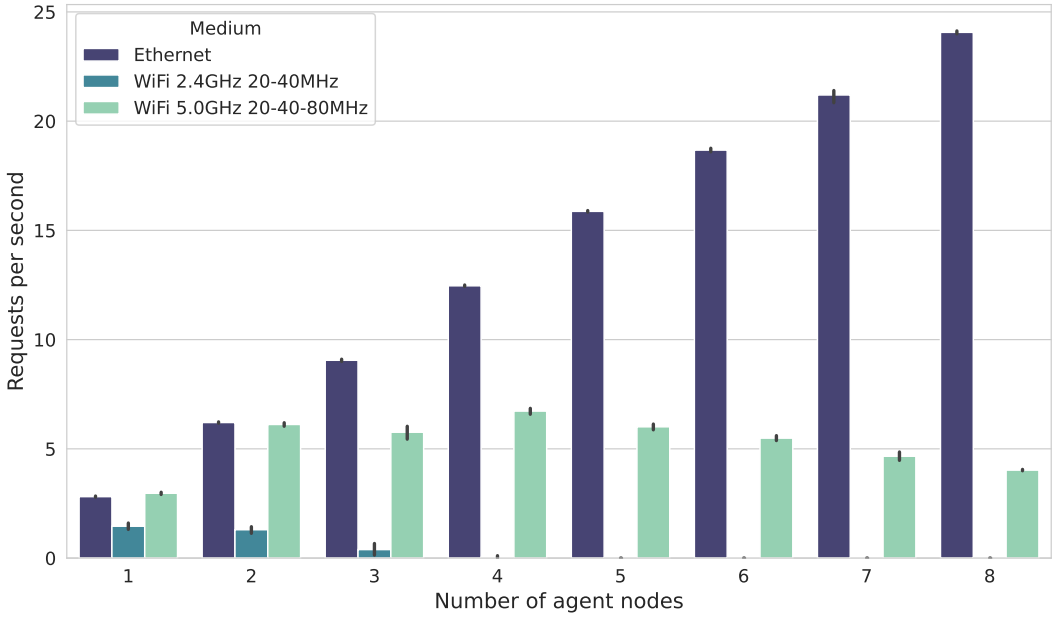

Figure 6.7: Evolution of the number of requests per second for an increasing number of agent nodes in the Raspberry Pi 5 cluster variant over different networking mediums.

6.4 Conclusion

Through the evaluation, we've seen that the clusters of upcycled smartphones built using the *K3S* framework show an improvement in performance with the number of agent nodes for Ethernet and Wi-Fi 5.0GHz. Clusters over Wi-Fi 2.4GHz show very poor performance. These were expected from the preliminary networking evaluation.

The results demonstrate the viability of such clusters, mainly over Ethernet, but Wi-Fi 5.0GHz may be a viable option for small-sized clusters since this medium does not require a switch.

6.5 Threats to validity and limitations

The results observed in this evaluation may be subject to several factors that influence positively or negatively the results. The measurements have not been made in a sanitized environment. The presence of other devices may have led to some additional interferences, reducing the performance over wireless mediums. Although it better represents a real-life use case, it makes the reproducibility of the experiment more difficult in a different environment.

It was also observed that the phone's power supply may have an impact on the performance. When powered up through its USB port, the device showed better performance than when powered only through the battery pin. We assume that this is due to a power-saving mode on the device to improve or not performance when it is plugged in or not.

In addition to the threats, the evolution has also some limitations. A main limitation is that all the experiments that required inference were done with the same image. While it shows the capability of the Fairphone 2 and cluster, the performance may change significantly for other image sizes and different workloads.

Chapter 7

Conclusion

In this thesis, we've demonstrated the possibility of creating a cluster of smartphones using the framework *K3S*. We've also shown that TensorFlow Lite could be used for image processing applications and run on devices with low computing power.

After a preliminary evaluation of the networking capabilities of the Fairphone 2 and Raspberry Pi 5, we determined that Ethernet would be more adapted for clusters with several connections. On wireless mediums, the devices show weaker results. Wi-Fi 5.0GHz could be viable as long as the number of connections stays low, but Wi-Fi 2.4GHz should be avoided because of the low goodput of the connections and the high risk of interference.

We've made a modular version of the image classification application provided by TensorFlow Lite with the capability of handling `HTTP POST` requests containing bitmap images using *Crow*. We characterized its global performance and determined the best parameters.

Finally, we've built two cluster variants using upcycled smartphones, one only composed of Fairphone 2 and another using a Raspberry Pi 5 for the server node. We configure it to deploy our application and analyze its performance. We've proven its viability for our application case over some networking mediums, such as Ethernet and Wi-Fi 5.0GHz. We also showed that Wi-Fi 2.4GHz was not adapted for this purpose at all.

We also discuss some threats of validity and limitations that may have influenced the results.

Future work

Complete device port Current port device port was still in testing with partial or missing functionalities. Even working functionalities may not be optimized. As the device port can be considered the foundation of any application that includes the device, making sure that it works correctly and that it is optimized would allow real development of the project using upcycled smartphones.

More recent devices This work used Fairphone 2 devices, which are already 9 years old. In addition to their poor specifications (only 2 GB of RAM), the device uses outdated technologies such as USB 2.0 while USB 3.0 is way more performant. Using slightly newer devices could significantly improve performance and provide enhanced support, such as enabling OpenCL support for the GPU.

A more advanced/complex application This work demonstrates the possibility of developing a simple cluster of upcycled smartphones using the *K3S* framework. We also deployed a rather basic and unoptimized TensorFlow Lite application on it. Now

that we've proved the possibility of such a thing, it could be interesting to create a more complex and advanced cluster and/or application that could be used in a real production environment.

Bibliography

[1] S. Moss, "From "brick" to smartphone: The evolution of the mobile phone," *MRS Bulletin*, vol. 46, no. 3, pp. 287–288, March 1, 2021, ISSN: 1938-1425. DOI: 10.1557/s43577-021-00067-7. [Online]. Available: https://doi.org/10.1557/s43577-021-00067-7 (visited on April 21, 2024).

[2] D. Hooi Ting, S. Fong Lim, T. Siuly Patanmacia, C. Gie Low, and G. Chuan Ker, "Dependency on smartphone and the impact on purchase behaviour," *Young Consumers*, vol. 12, no. 3, pp. 193–203, January 1, 2011, Publisher: Emerald Group Publishing Limited, ISSN: 1747-3616. DOI: 10.1108/17473611111163250. [Online]. Available: https://doi.org/10.1108/17473611111163250 (visited on Jul. 26, 2024).

[3] D. Watson, A. C. Gylling, N. Tojo, H. Throne-Holst, B. Bauer, and L. Milios, *Circular Business Models in the Mobile Phone Industry*. Nordic Council of Ministers, November 30, 2017, 90 pp., Google-Books-ID: cYY9DwAAQBAJ, ISBN: 978-92-893-5203-1.

[4] E. P. Agency. "Waste electrical and electronic equipment (WEEE)." (), [Online]. Available: https://www.epa.ie/our-services/compliance--enforcement/waste/weee/ (visited on May 20, 2024).

[5] P. Sarath, S. Bonda, S. Mohanty, and S. K. Nayak, "Mobile phone waste management and recycling: Views and trends," *Waste Management*, vol. 46, pp. 536–545, December 1, 2015, ISSN: 0956-053X. DOI: 10.1016/j.wasman.2015.09.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0956053X15301227 (visited on May 20, 2024).

[6] "IEEE standard for environmental assessment of televisions," *IEEE Std 1680.3-2012*, pp. 1–61, October 2012, Conference Name: IEEE Std 1680.3-2012. DOI: 10.1109/IEEESTD.2012.6331499. [Online]. Available: https://ieeexplore.ieee.org/document/6331499 (visited on May 20, 2024).

[7] T. Makov and C. Fitzpatrick, "Is repairability enough? big data insights into smartphone obsolescence and consumer interest in repair," *Journal of Cleaner Production*, vol. 313, p. 127 561, Sep. 1, 2021, ISSN: 0959-6526. DOI: 10.1016/j.jclepro.2021.127561. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0959652621017790 (visited on May 20, 2024).

[8] X. Li, P. J. Ortiz, J. Browne, *et al.*, "A case for smartphone reuse to augment elementary school education," in *International Conference on Green Computing*, August 2010, pp. 459–466. DOI: 10.1109/GREENCOMP.2010.5598279. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5598279 (visited on April 21, 2024).

[9] S. Ward and M. Gittens, "Building useful smart campus applications using a retired cell phone repurposing model," in *2018 Third International Conference on Electrical and Biomedical Engineering, Clean Energy and Green Computing (EBECEGC)*, April 2018, pp. 43–48. DOI: 10.1109/EBECEGC.2018.8357131. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8357131 (visited on April 21, 2024).

[10] J. Chan, A. Raghunath, K. E. Michaelsen, and S. Gollakota, "Testing a drop of liquid using smartphone LiDAR," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 6, no. 1, 3:1–3:27, March 29, 2022. DOI: 10.1145/3517256. [Online]. Available: https://dl.acm.org/doi/10.1145/3517256 (visited on April 21, 2024).

[11] D. Whiteson, M. Mulhearn, C. Shimmin, K. Cranmer, K. Brodie, and D. Burns, "Searching for ultra-high energy cosmic rays with smartphones," *Astroparticle Physics*, vol. 79, pp. 1–9, Jun. 1, 2016, ISSN: 0927-6505. DOI: 10.1016/j.astropartphys.2016.02.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0927650516300147 (visited on April 21, 2024).

[12] B. Na, J. Jang, S. Park, *et al.* "Scalable smartphone cluster for deep learning," arXiv.org. (October 23, 2021), [Online]. Available: https://arxiv.org/abs/2110.12172v1 (visited on October 10, 2023).

[13] D. Marković, D. Vujičić, D. Mitrović, and S. Ranđić, "Image processing on raspberry pi cluster," *International Journal of Electrical Engineering and Computing*, vol. 2, no. 2, pp. 83–90, December 27, 2018, Number: 2, ISSN: 2566-3682. DOI: 10.7251/IJEEC1802083M. [Online]. Available: https://ijeec.etf.ues.rs.ba/index.php/ijeec/article/view/38 (visited on November 8, 2023).

[14] J. Switzer, E. Siu, S. Ramesh, R. Hu, and E. Zadorian, "Data centers from discarded cell phones," 2021. [Online]. Available: https://www.semanticscholar.org/paper/Data-Centers-from-Discarded-Cell-Phones-Switzer-Siu/e1a3aa6826a6e1705480c4d65ae4da74ed6c465f (visited on October 10, 2023).

[15] J. Switzer, E. Siu, S. Ramesh, R. Hu, E. Zadorian, and R. Kastner, "Renée: New life for old phones," *IEEE Embedded Systems Letters*, vol. 14, no. 3, pp. 135–138, Sep. 2022, Conference Name: IEEE Embedded Systems Letters, ISSN: 1943-0671. DOI: 10.1109/LES.2022.3147409. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9695989 (visited on October 10, 2023).

[16] J. Switzer, G. Marcano, R. Kastner, and P. Pannuto, "Junkyard computing: Repurposing discarded smartphones to minimize carbon," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023, New York, NY, USA: Association for Computing Machinery, January 30, 2023, pp. 400–412, ISBN: 978-1-4503-9916-6. DOI: 10.1145/3575693.3575710. [Online]. Available: https://dl.acm.org/doi/10.1145/3575693.3575710 (visited on October 10, 2023).

[17] S. Harizopoulos and S. Papadimitriou, "A case for micro-cellstores: Energy-efficient data management on recycled smartphones," in *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, ser. DaMoN '11, New York, NY, USA: Association for Computing Machinery, Jun. 13, 2011, pp. 50–55, ISBN: 978-1-4503-0658-4. DOI: 10.1145/1995441.1995448. [Online]. Available: https://doi.org/10.1145/1995441.1995448 (visited on April 21, 2024).

[18] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE Access*, vol. 8, pp. 85 714–85 728, 2020, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2991734. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9083958 (visited on Jul. 31, 2024).

[19] J. Gonzalez, J. Hunt, M. Thomas, R. Anderson, and U. Mangla. "Edge computing architecture and use cases." (April 7, 2022), [Online]. Available: https://developer.ibm.com/articles/edge-computing-architecture-and-use-cases/#cross-industry-use-cases9 (visited on Jul. 14, 2024).

[20] "Ubuntu Touch." (), [Online]. Available: https://ubuntu-touch.io/ (visited on August 14, 2024).

[21] "postmarketOS." (2024), [Online]. Available: https://postmarketos.org/ (visited on August 14, 2024).

[22] postmarketOS. "About postmarketOS." (April 2024), [Online]. Available: https://postmarketos.org/ (visited on August 14, 2024).

[23] postmarketOS. "(Close to) Mainline." (Jul. 2023), [Online]. Available: https://wiki.postmarketos.org/wiki/(Close_to)_Mainline (visited on August 14, 2024).

[24] "Alpine Linux." (2024), [Online]. Available: https://alpinelinux.org/ (visited on August 14, 2024).

[25] "musl libc." (), [Online]. Available: https://musl.libc.org/ (visited on August 14, 2024).

[26] "busybox." (2023), [Online]. Available: https://busybox.net/ (visited on August 14, 2024).

[27] "The GNU C Library." (), [Online]. Available: https://www.gnu.org/software/libc/ (visited on August 14, 2024).

[28] "TensorFlow." (), [Online]. Available: https://www.tensorflow.org/ (visited on August 14, 2024).

[29] "TensorFlow Lite." (), [Online]. Available: https://www.tensorflow.org/lite (visited on August 14, 2024).

[30] "Fairphone." (), [Online]. Available: https://www.fairphone.com/ (visited on August 14, 2024).

[31] "Fairphone 2 FAQ." (March 2022), [Online]. Available: https://support.fairphone.com/hc/en-us/articles/4608508653713-Fairphone-2-FAQ (visited on August 14, 2024).

[32] "Swarn." (2024), [Online]. Available: https://swarn.be/ (visited on Jul. 31, 2024).

[33] *postmarketOS: Fairphone 2 (fairphone-fp2)*, Jun. 2023. [Online]. Available: https://wiki.postmarketos.org/wiki/Fairphone_2_(fairphone-fp2) (visited on August 19, 2024).

[34] "Raspberry Pi 5." (), [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-5/ (visited on August 14, 2024).

[35] *Mikrotik cAP ac*. [Online]. Available: https://mikrotik.com/product/cap_ac (visited on August 19, 2024).

[36] *HP ProBook 450 G6 Notebook PC Specifications*. [Online]. Available: https://support.hp.com/sg-en/document/c06191146 (visited on August 19, 2024).

[37] "iperf." (2024), [Online]. Available: https://iperf.fr/ (visited on August 1, 2024).

[38] *ping(8) - Linux man page*. [Online]. Available: https://linux.die.net/man/8/ping (visited on August 14, 2024).

[39] "Error initializing Wifi 40Mhz channel width in drivers/net/wireless/broadcom/brcm80211/brcmfm (2024), [Online]. Available: https://github.com/raspberrypi/linux/issues/3415 (visited on August 13, 2024).

[40] "Pi 4 WiFi fails to use 40MHz channels on 2.4GHz." (Sep. 2023), [Online]. Available: https://forums.raspberrypi.com/viewtopic.php?t=356752 (visited on August 13, 2024).

[41] *Multiple port match support*. [Online]. Available: https://www.kernelconfig.io/config_netfilter_xt_match_multiport (visited on August 14, 2024).

[42] *K3S: Requirements*, August 2024. [Online]. Available: https://docs.k3s.io/installation/requirements (visited on August 14, 2024).

[43] *K3S: Architecture: Single-server Setup with an Embedded DB*. [Online]. Available: https://docs.k3s.io/architecture#single-server-setup-with-an-embedded-db (visited on August 17, 2024).

[44] *TensorFlow: Image classification*, August 2023. [Online]. Available: https://www.tensorflow.org/lite/examples/image_classification/overview (visited on August 18, 2024).

[45] *Github: wrk*. [Online]. Available: https://github.com/wg/wrk (visited on August 19, 2024).

[46] *Wikipedia: Grace Hopper*, August 2024. [Online]. Available: https://en.wikipedia.org/wiki/Grace_Hopper (visited on August 19, 2024).

Appendix A

Dockerfile to build pod image

The Dockerfile displayed in listing 4, was used to build the image run in the container of the pod of the deployed application. It runs, the Crow version of the TensorFlow Lite application implemented in this work for validation.

```
FROM alpine:latest
COPY ./label_image_with_crow_app /app
WORKDIR /app
RUN apk update && \
    apk add \
        libstdc++ \
        bash \
        nano

EXPOSE 18080
ENTRYPOINT ["./label_image_with_crow"]
CMD ["-c", "config.json"]
```

Listing 4: `Dockerfile` use to build the container image run inside K3S pod of the application deployment.

Appendix B

Crow: HTTP POST request handling function

The implementation of the `Crow` web service function handling images from HTTP requests.

```
crow::response handle_image(const crow::request& req, std::string& file) {
    // Check if request's content-type is multipart
    const std::string& req_content_type = req.get_header_value("Content-Type");
    const std::string& multipart_content_type = "multipart/form-data";
    if (req_content_type.find(multipart_content_type) == std::string::npos) {
        return crow::response(crow::status::BAD_REQUEST,
                    "Invalid content type: multipart/form-data expected.\n");
    }
    // Check if the request contains an image
    crow::multipart::message msg(req);
    crow::multipart::part image_part = msg.get_part_by_name("image");
    std::string img_content_type = image_part
                                    .get_header_object("Content-Type").value;
    const std::string& octet_stream_content_type = "application/octet-stream";
    if (img_content_type.find(octet_stream_content_type) == std::string::npos) {
        return crow::response(crow::status::BAD_REQUEST, "Missing image data.\n");
    }
    // Save the bitmap image locally
    std::string image_data = image_part.body;
    std::ofstream out(file, std::ios::out | std::ios::binary);
    if (!out.is_open()) {
        return crow::response(crow::status::INTERNAL_SERVER_ERROR,
                    "Could not save the image: output file not opening.\n");
    }
    out.write(image_data.c_str(), image_data.length());
    out.close();
    return crow::response(crow::status::OK);
}
```

Listing 5: Implementation of the Crow function handling HTTP POST requests with image.

Appendix C

TensorFlow Lite application evaluation: Execution time results for wireless mediums

This annex regroups additional measures of the first experiment on execution time by subparts and parts for the two wireless mediums, i.e. Wi-Fi 2.4GHz and Wi-Fi 5.0GHz.

Wi-Fi 2.4GHz

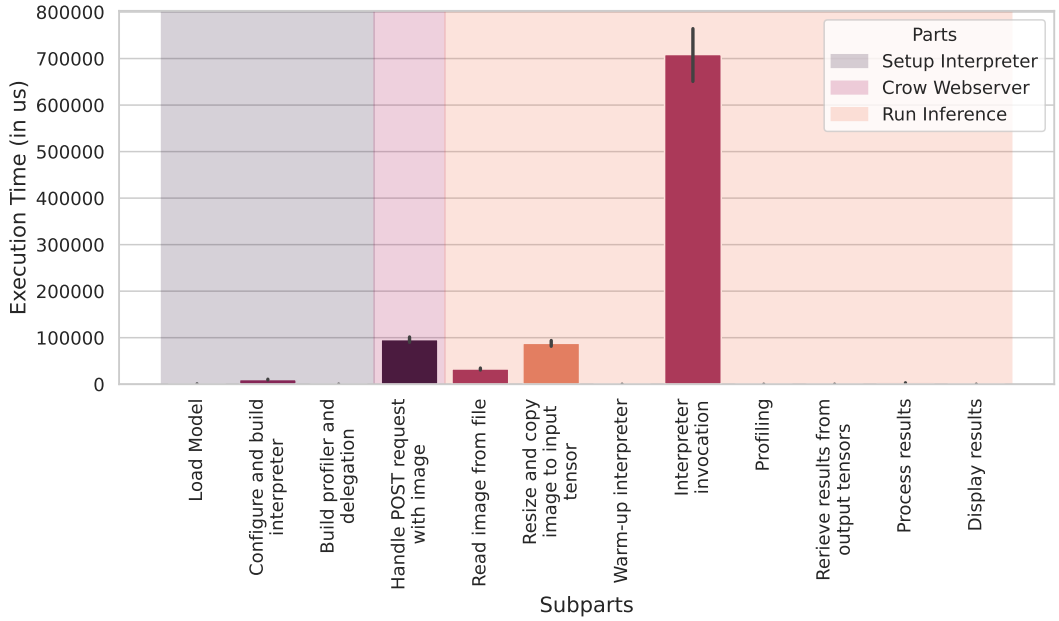

Figure C.1: Comparison between the execution time in microseconds of each subpart of the TensorFlow Lite image classification application over the Wi-Fi 2.4GHz. Subparts have been grouped according to the part of the application to which they belong and have been sorted in execution order.

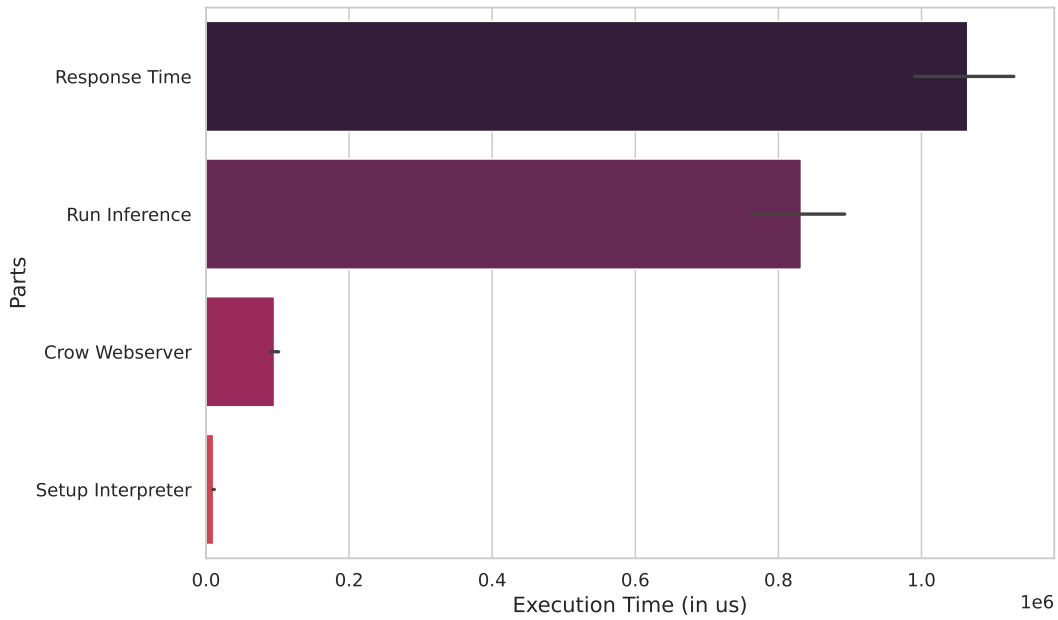

Figure C.2: Comparison between the execution time in microseconds of each part of the TensorFlow Lite image classification application and the total execution time, over Wi-Fi 2.4GHz.

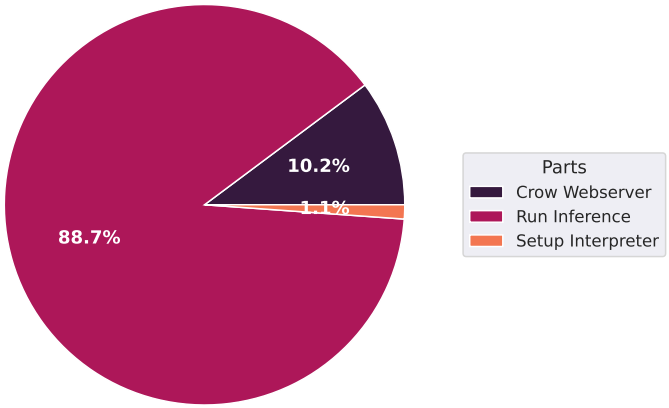

Figure C.3: Ratio of the total execution time for each part of the TensorFlow Lite image classification application over Wi-Fi 2.4GHz

Wi-Fi 5.0GHz

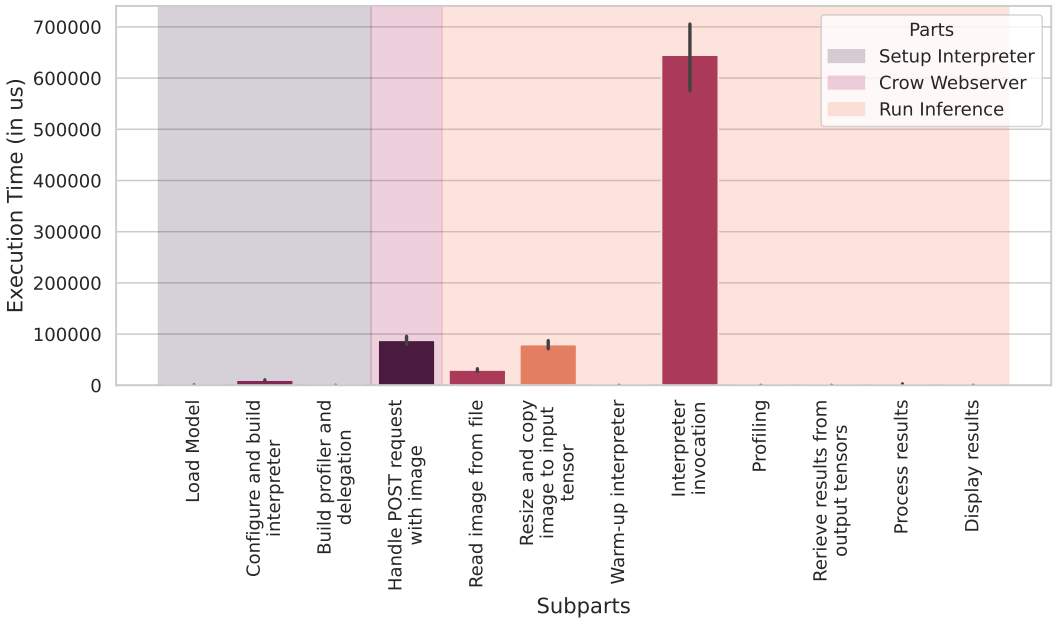

Figure C.4: Comparison between the execution time in microseconds of each subpart of the TensorFlow Lite image classification application over the Wi-Fi 5.0GHz. Subparts have been grouped according to the part of the application to which they belong and have been sorted in execution order.

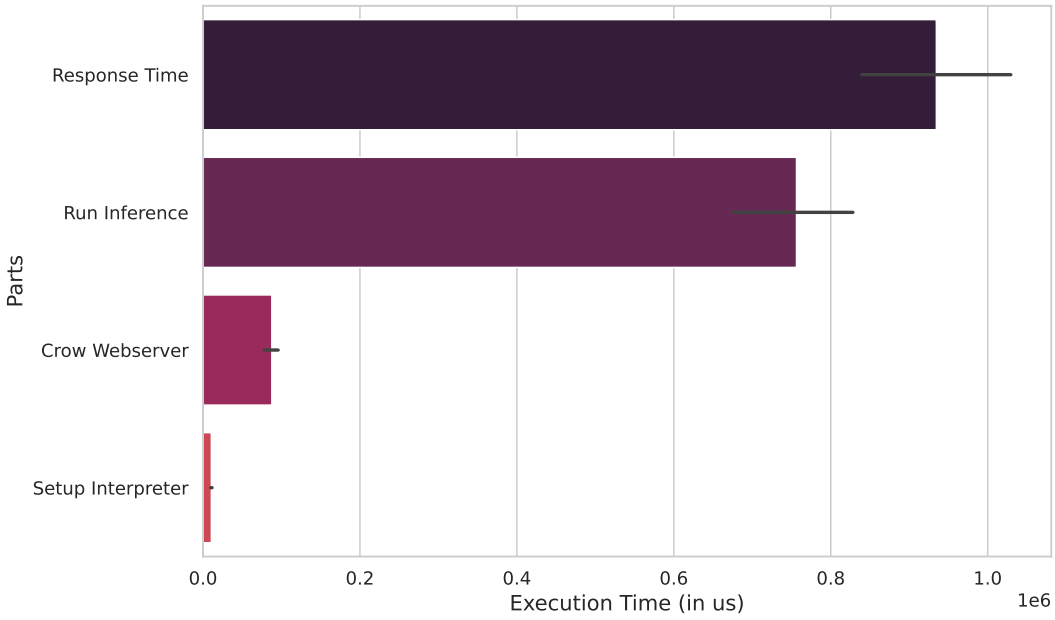

Figure C.5: Comparison between the execution time in microseconds of each part of the TensorFlow Lite image classification application and the total execution time, over Wi-Fi 5.0GHz.

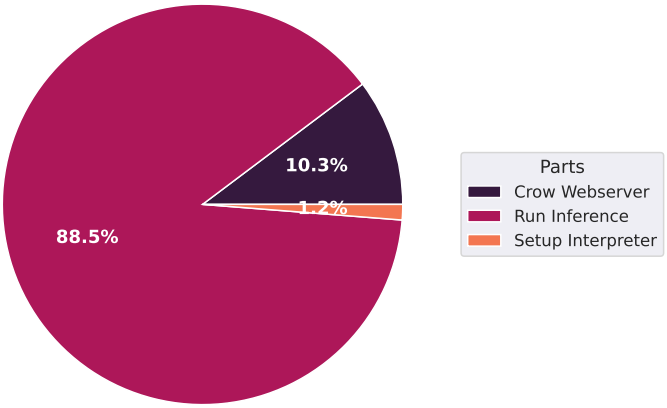

Figure C.6: Ratio of the total execution time for each part of the TensorFlow Lite image classification application over Wi-Fi 5.0Hz

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**