

APPROVAL

## D2.3

# Enhanced IoT Underlying Technology (Final Version)

DRAFT

**WORKPACKAGE** WP2

**DOCUMENT** D2.3

**REVISION** V0.3

**DELIVERY DATE** 31/03/2023

**PROGRAMME IDENTIFIER** H2020-ICT-2020-1

**GRANT AGREEMENT ID** 957246

**START DATE OF THE PROJECT** 01/10/2020

**DURATION** 3 YEARS

© Copyright by the IoT-NGIN Consortium

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 957246



## DISCLAIMER

This document does not represent the opinion of the European Commission, and the European Commission is not responsible for any use that might be made of its content.

This document may contain material, which is the copyright of certain IoT-NGIN consortium parties, and may not be reproduced or copied without permission. All IoT-NGIN consortium parties have agreed to full publication of this document. The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the IoT-NGIN consortium as a whole, nor a certain party of the IoT-NGIN consortium warrant that the information contained in this document is capable of use, nor that use of the information is free from risk, and does not accept any liability for loss or damage suffered using this information.

## ACKNOWLEDGEMENT

This document is a deliverable of IoT-NGIN project. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 957246.

The opinions expressed in this document reflect only the author's view and in no way reflect the European Commission's opinions. The European Commission is not responsible for any use that may be made of the information it contains.

<b>PROJECT ACRONYM</b>	IoT-NGIN
<b>PROJECT TITLE</b>	Next Generation IoT as part of Next Generation Internet
<b>CALL ID</b>	H2020-ICT-2020-1
<b>CALL NAME</b>	Information and Communication Technologies
<b>TOPIC</b>	ICT-56-2020 - Next Generation Internet of Things
<b>TYPE OF ACTION</b>	Research and Innovation Action
<b>COORDINATOR</b>	Capgemini Technology Services (CAP)
<b>PRINCIPAL CONTRACTORS</b>	Atos Spain S.A. (ATOS), ERICSSON GmbH (EDD), ABB Oy (ABB), NETCOMPANY-INTRASOFT SA (INTRA), Engineering-Ingegneria Informatica SPA (ENG), Robert Bosch Espana Fabrica Aranjuez SA (BOSCHN), ASM Terni SpA (ASM), Forum Virium Helsinki (FVH), ENTERSOFT SA (OPT), eBOS Technologies Ltd (EBOS), Privanova SAS (PRI), Synelxis Solutions S.A. (SYN), CUMUCORE Oy (CMC), Emotion s.r.l. (EMOT), AALTO-Korkeakoulusaatio (AALTO), i2CAT Foundation (I2CAT), Rheinisch-Westfälische Technische Hochschule Aachen (RWTH), Sorbonne Université (SU)
<b>WORKPACKAGE</b>	WP2
<b>DELIVERABLE TYPE</b>	REPORT
<b>DISSEMINATION LEVEL</b>	PUBLIC
<b>DELIVERABLE STATE</b>	FINAL
<b>CONTRACTUAL DATE OF DELIVERY</b>	31/03/2023
<b>ACTUAL DATE OF DELIVERY</b>	03/04/2023
<b>DOCUMENT TITLE</b>	Enhanced IoT Underlying Technology (Final Version)
<b>AUTHOR(S)</b>	Jonathan Klimt (RWTH), Stefan Lankes (RWTH), Manuel Pitz (RWTH), Jesús Gorroñoigoitia (ATOS), Tomas Lagos (SU), Serge Fdida (SU), Marcelo Dias de Amorim (SU), Marios Sophocleous (EBOS), Jose Costa-Requena (CMC)
<b>REVIEWER(S)</b>	Serge Fdida (SU), Tomas Lagos (SU), Jose Costa-Requena (CMC)
<b>ABSTRACT</b>	SEE EXECUTIVE SUMMARY
<b>HISTORY</b>	SEE DOCUMENT HISTORY
<b>KEYWORDS</b>	IoT, 5G, API, Unikernel, Enhancement, Device-to-Device, D2D, 5G-LAN, TSN

## Document History

Version	Date	Contributor(s)	Description
V0.1	09/01/2023	RWTH	First draft
V0.2	27/03/2023	SU, CMC, EBOS, RWTH, ATOS	Internal review version
V0.3	31/03/2023	RWTH, SU, CMC	Final version

DRAFT - PENDING EC APPROVAL

# Table of Contents

Document History .....	4
Table of Contents .....	5
List of Figures.....	7
List of Tables.....	8
List of Acronyms and Abbreviations.....	9
Executive Summary .....	11
1 Introduction.....	12
1.1 Intended Audience .....	12
1.2 Relations to other activities.....	12
1.3 Document overview.....	12
2 Enhancing 5G functionality to improve 5G coverage .....	13
2.1 Introduction .....	13
2.2 Experimental setup .....	14
2.3 Experimental methodology.....	15
2.4 Experimental Results .....	16
2.4.1 5G network .....	16
2.4.2 WiFi Direct Network.....	21
2.4.3 End-to-End D2D Communication Testing.....	24
2.4.4 Relay selection.....	25
2.5 Conclusions.....	26
3 Enabling 5G to support protocols for deterministic communications.....	27
3.1 Introduction .....	28
3.2 Results.....	29
3.3 Conclusions.....	30
4 Enhancing 5G ease of use through improved 5G APIs .....	32
4.1 Introduction .....	32
4.2 Final API .....	32
4.3 Implementation.....	36
4.3.1 IoT-NGIN API Server.....	36
4.3.2 Unikernel Adapter.....	36
4.3.3 Slice Management Adapter.....	37
4.3.4 Slice Management simulator.....	37
4.4 Results .....	38

4.4.1	API Server .....	38
4.4.2	Unikernel Demo.....	38
4.4.3	Slice Manager Demo .....	39
4.5	Conclusions.....	40
5	Secure Edge Cloud Framework for micro-services .....	41
5.1	Introduction .....	41
5.2	The IoT-NGIN Secure Edge Cloud Framework .....	44
5.3	Application to ML optimization in IoT-NGIN .....	45
5.3.1	Machine learning support using TVM cross compilation .....	47
5.3.2	Torch C++ ML Models in Unikernels .....	47
5.4	Conclusions.....	50
6	How the technology is transferred and demonstrated in use cases and living labs .....	51
7	Conclusions.....	52

DRAFT - PENDING EC APPROVAL

## List of Figures

Figure 2-1: Overall architecture of the experimental setup. ....	14
Figure 2-2: Test between 5GC Core and Relay device. ....	19
Figure 2-3: Testing of Wi-Fi Direct Network.....	22
Figure 2-4: End-to-End experimental setup.....	24
Figure 2-5: RTT times versus number of measurements for End-to-End tests.....	25
Figure 2-6: Architecture for relay selection. ....	26
Figure 3-1: TSN network deployed at ABB living labs for time synchronization testing. ....	28
Figure 3-2: Equipment deployed at ABB living labs for time synchronization testing. ....	30
Figure 4-1 IoT-NGIN 5G resource management API.....	32
Figure 4-2: Open API UI .....	34
Figure 4-3: OpenAPI visualization of the <code>get_service</code> call's response.....	35
Figure 4-4: Call and response of <code>start_service</code> in the command line. ....	35
Figure 4-5: The IoT-NGIN 5G Resource Management API server in action.....	38
Figure 4-6: Unikernel application started with IoT NGIN 5G API. ....	38
Figure 4-7: Rancher UI after starting a Unikernel hello world service.....	39
Figure 4-8: Slice Manager command line tool.....	39
Figure 4-9: curl command to start a service.....	40
Figure 4-10: OpenAPI visualization of the <code>start_service</code> call. ....	40
Figure 5-1: Classical micro-service stacks. Containerization on the left side, virtual machines on the right side. ....	41
Figure 5-2: Hardened container setup using containers, microVMs and library operating systems. ....	42
Figure 5-3: Rusty-Hermit conceptual overview .....	43
Figure 5-4: The integration of runh in the Docker and Kubernetes software stack .....	43
Figure 5-5: Performance comparison of runh and runc using QEMU and microVMs.....	44

## List of Tables

Table 2-1: RTT from gNB to Relay device.....	16
Table 2-2: RTT from Relay to gNB .....	16
Table 2-3: Measure the Throughput from gNB to Relay (R – Reverse mode).....	17
Table 2-4: Measure the Throughput from Relay to gNB .....	17
Table 2-5: Measure the Throughput from Relay to gNB with PLMN 999-99.....	18
Table 2-6: Measurements of Jitter between from 5GC Core to Relay.....	18
Table 2-7: Measurements of Jitter between from Relay to 5GC Core.....	19
Table 2-8: RTT from 5GC Core to Relay device .....	20
Table 2-9: RTT from Relay to 5GC Core.....	20
Table 2-10: Measure the Throughput from 5GC Core to Relay (R – Reverse mode).....	20
Table 2-11: Measure the Throughput from Relay to 5GC. ....	20
Table 2-12: Measure the Jitter between from 5GC Core to Relay .....	21
Table 2-13: Measure the Jitter between from Relay to 5GC Core .....	21
Table 2-14: Measure the RTT from End-Device to Relay.....	22
Table 2-15: Measure the RTT from Relay to End-Device.....	23
Table 2-16: Measure the Throughput from End-Device to Relay.....	23
Table 2-17: Measure the Throughput from Relay to End-Device.....	23
Table 3-1: Measure the time synchronization in TSN device connected to UE .....	29
Table 3-2: Measure the time synchronization in fixed TSN device .....	29
Table 4-1: API calls .....	33
Table 4-2: Unikernel Adapter API calls .....	36
Table 4-3: I2CAT Backend API calls. ....	37
Table 4-4: Slice Management Simulator calls.....	37



## List of Acronyms and Abbreviations

<b>IoT</b>	Internet of Things
<b>WP</b>	Work Package
<b>3GPP</b>	3rd Generation Partnership Project
<b>5G</b>	Fifth Generation (mobile network)
<b>5G NR</b>	5G New Radio
<b>5GC</b>	5G Core Network
<b>5GS</b>	5G System
<b>AF</b>	Application Function
<b>API</b>	Application Programming Interface
<b>BS</b>	Base Station
<b>CB</b>	Frame Preemption
<b>CPU</b>	Central Processing Unit
<b>D2D</b>	Device-to-Device
<b>DNN</b>	Data Network Name
<b>DS-TT</b>	Device Synchronization Time Transfer
<b>EMI</b>	Electro-Magnetic Interference
<b>FPGAs</b>	Field-Programmable Gate Arrays
<b>GM</b>	Grand Master
<b>gNB</b>	5G New Radio Base Station
<b>gPTP</b>	Generalized Precision Time Protocol
<b>HTTP</b>	Hypertext Transfer Protocol
<b>HTTPS</b>	Hypertext Transfer Protocol Secure
<b>ICs</b>	Integrated Circuits
<b>IEEE</b>	Institute of Electrical and Electronics Engineers
<b>IP</b>	Internet Protocol
<b>IoT</b>	Internet of Things
<b>LAN</b>	Local Area Network

<b>LLDP</b>	Link Layer Discovery Protocol
<b>MAC</b>	Medium Access Control
<b>ML</b>	Machine Learning
<b>mMTC</b>	Massive Machine-Type Communications
<b>NF</b>	Network Function
<b>NMS</b>	Network Management System
<b>NPN</b>	Non-Public Networks
<b>NR</b>	New Radio
<b>OAM</b>	Operations, Administration, and Maintenance
<b>OCI</b>	Open Container Initiative
<b>OS</b>	Operating System
<b>PDU</b>	Protocol Data Unit
<b>POSIX</b>	Portable Operating System Interface
<b>PSFP</b>	Per-Stream Filtering and Policing
<b>ProSe</b>	Proximity Services
<b>QEMU</b>	Quick EMUlator
<b>Qbu</b>	Frame Replication and Elimination for Reliability
<b>Qbv</b>	Enhancements for Scheduled Traffic
<b>QoS</b>	Quality of Service
<b>RTT</b>	Round-Trip Time
<b>SA</b>	Standalone
<b>TSN</b>	Time-Sensitive Networking
<b>UC</b>	Use Case
<b>UE</b>	User Equipment
<b>UI</b>	User Interface
<b>V2X</b>	Vehicle-to-Everything
<b>VM</b>	Virtual Machine
<b>VN</b>	Virtual Network

## Executive Summary

This deliverable is the third and final deliverable concerning the work for Enhancing IoT Underlying Technology that presents the outcomes and results of the work conducted. The four main outcomes described are:

### **Improved 5G coverage using D2D connections**

This chapter describes in detail the laboratory setup and a variety of tests conducted to validate the possibility of extending the 5G coverage through 5G devices. The tests conducted cover the topics of coverage, throughput and latency. The analysis is carried out in different configurations of user equipment (UE) and base stations.

The tests were performed using relays and mobile phones, and the measurements were taken between the 5G network core (5GC), a base station (BS), and relay devices. More specifically, the measurement of the RTT in different sections of the laboratory setup. The results show that the measured RTT times are within the acceptable range for 5G SA eMBB according to 3GPP, Release 16 standards. Furthermore, the tests show that D2D communication in 5G networks is feasible with adequate coverage. The experiments demonstrate that coverage can be improved by placing D2D relay devices near the BS station for higher performance. However, external factors such as Electro-Magnetic Interference (EMI) can affect the network's performance.

### **Deterministic and time sensitive communication for 5G**

In this chapter the test setup for timesynchronization for a private 5G network is described. This includes the deployment at a living lab trial site. The measurements conducted show the time difference between a grand master clock and the local time of a UE.

The 5G core with TSN functionality was deployed and tested in a control environment at CMC laboratory as well as in ABB living labs. The results showed the time synchronization was achieved both using IP over VxLAN and native synchronization protocol over Ethernet PDU session. However, the Ethernet PDU is not fully supported in devices and base stations to complete synchronization requires additional equipment updates.

### **Simplified 5G resources management API**

The final proposed API is described and the server architecture as well as various adapters for existing infrastructure components are explained. The functionality is evaluated either by demonstrating the API use with real backends like a kubernetes instance, or by running it against simulated backends. The API is also tested against parts of the secure edge cloud framework described in the last chapter.

### **Secure edge cloud framework**

The final version of the secure edge cloud framework is described. The framework combines unikernels, the orchestration tool kubernetes and multiple ways of executing ML inference in said technologies. This does not only increase the isolation of applications, it can also reduce the memory footprint and faster startup times can also increase the performance. By running ML models from the project with the framework's technologies, the usability as well as the performance is demonstrated.

# 1 Introduction

Since the publication of our first description of the 5G enhancements in D2.1 in November, 2021 and this document, a lot of work was put in the technologies developed for the enhancement for future internet and network technologies. Our work has progressed according to plan, so we can present the final results in this deliverable.

This first chapter is intended to provide information necessary to enable the reader to understand the structure of this deliverable and how the individual chapters relate to each other. This chapter relates the contents of this deliverable to other existing project deliverables and to the future deliverables of the IoT-NGIN project.

## 1.1 Intended Audience

This deliverable will be useful to a wide audience of readers. As it provides an in-depth description of the outcomes and achievements on 5G enhancements, it is useful for project reviewers and the European Commission as well as project partners that want to understand the details of the technology better.

The enhancements are mostly related to industrial 5G applications and backend technologies. Thus, this document is also relevant for readers in the IoT, mobile communications, smart agriculture, smart industry, smart cities and smart energy sectors as we see the most potential for new products, services and use cases in these domains in the coming years.

## 1.2 Relations to other activities

For more information the previous deliverables on this topic can be consulted. These are deliverable *D2.1 Enhancing IoT M2M/MCM Communications* and *D2.2 Enhancing IoT Underlying Technology*.

## 1.3 Document overview

In this document, we give room for each of the enhancements developed. This leaves us with the following structure:

- A presentation of the connectivity enhancements by utilizing device-to-device communications is given in chapter 2.
- Deterministic communications using time sensitive communications and 5GLAN are discussed in chapter 3.
- Chapter 4 presents the 5G Resources Management API, which simplifies the use of various 5G functionalities.
- The last enhancement - the Secure Execution Environment for Edge Cloud Services - is presented in chapter 5.
- A demonstration of the technology transfer in the living labs is given in chapter 6.
- The document is concluded with a short summary in chapter 7.

## 2 Enhancing 5G functionality to improve 5G coverage

This first chapter presents the results and outcomes of the work that has been done to improve mobile connectivity.

### 2.1 Introduction

Cellular networks are being developed with the ambition to cover most of user needs, extending their suitability by standardizing new generations over time, e.g., 4G, 5G and now 6G. However, during this journey, other technologies and/or architectures are considered to cover specific needs. The current networking architecture is converging with the best practices that constitute the substrate for the cloud, adopting a very centralised approach (cloud-native) distributed using the edge concept. However, there are always situations when cellular does not apply. It could be either because of limited or transient coverage or when there is no coverage due to failure, damage or too costly deployment.

Device-to-device (D2D) is a specific solution considered in the past with limited success. However, as technology and knowledge evolve, it is now considered again in various environments. In 3GPP (3rd Generation Partnership Project), research has been conducted on various aspects of D2D communication, including its potential to improve network capacity and coverage and support new use cases, such as public safety and massive machine-type communications (mMTC). D2D communication in 3GPP has been standardised in Release 12 under the terminology "Proximity Services" (ProSe). It focuses on integrating D2D communication into existing cellular networks to support services such as Discovery, Emergency, Content Sharing, Location Privacy, and others. In addition, it is being developed in releases 17 and 18, covering a range of services based on proximity, such as V2X.

As ProSe is developing slowly, intermediate solutions exploiting and combining various technologies similar to 5G and Wi-Fi/Bluetooth are needed. We propose, study and develop a simple but effective methodology for coverage extension by establishing a D2D communication between the node outside the cell coverage area and a relay inside the coverage area. A metrics exchange is established between the participants to select the most suitable relay for the target performance. This selection process exploits various metrics of interest that can be enriched if necessary. This fits perfectly into the IoT-NGIN context, where one challenge related to deploying 5G in wide area networks is the short network coverage of approximately 500 m depending on the frequencies used and the attenuation of signals in a given location. In applications such as smart cities (UC1 of IoT-NGIN), a high density of base stations is required for complete coverage due to the short coverage range. Therefore, IoT-NGIN proposes to use D2D communications to enhance coverage in areas not well served by public networks.

A solution addressing this particular need has been designed and tested. The relay selection process has been architected using extensive analysis of the D2D link properties in various conditions. Our AtomD tool has been released for that purpose and is available as an open-source component. A second solution has been considered, which is based on a 5G LAN technology. The 5G LAN consists of a LAN network where all connected devices to the same network use the same 5G Core and slice. As a result, the devices are visible to each other.

The concept here is that each device and relay connected to the 5G network will also be registered in a 5G LAN created by the 5G Core.

This setup has been instrumented and analysed to capture the system's performance and diverse configurations and parameters. The main outcome is presented in the next section.

## 2.2 Experimental setup

This section presents the experimental setup, where we establish an end-to-end environment involving 4 UEs. To do so, we use 2 OnePlus 8T for the end-devices situated outside a cellular coverage range and 2 Nokia XR20 inside a cellular coverage range. It is important to highlight that these phones have Qualcomm SM4350 Snapdragon 480 5G.

Given each device's location, we establish that the Nokia phones are used as Relay devices. In addition, we established a laboratory-based Base Station (BS) that used the Amarisoft Callbox Mini system with a gNodeB (gNB) compliant with 3GPP Release 16, which uses 20 MHz bandwidth. This BS is a desktop PC with integrated SDR card with limited specifications. The maximum speed is estimated at around 200 Mbps. Moreover, according to the provider, the system's wireless range is around 10 metres. In addition, we integrate a 5G Core (5GC) developed by CUMUCORE, which runs on a Linux-based PC and a custom-made mobile/server application running on a Raspberry Pi 400, where we implemented an application that manages the connections between the relay devices and end-devices.

We use custom SIM cards to connect to the BS and establish communication with the 5GC. Here, the 5GC provides the required settings for the SIM cards. Some essential parameters for SIM cards are the Public Land Mobile Network (PLMN), international mobile subscriber identity (IMSI) number and secret key for provider and user. PLMN-00101 is for designed to be used for a testing network and 99999 is designed for private networks. The connection between the BS, 5GC and the Raspberry Pi was achieved through Ethernet connections to a 1 Gbps router (MikroTik RouterOS). This system is developed on the local private network to reduce external noise. The gNB and Relay devices were connected with 5G New Radio (NR), in the Stand-Alone (SA) mode. The End-Devices and Relay devices were connected with Wi-Fi direct through the developed mobile/server application.

The overall experimental setup and configuration are shown in Figure 2-1. This configuration was also used to test the relay selection capability of the mobile/server application.

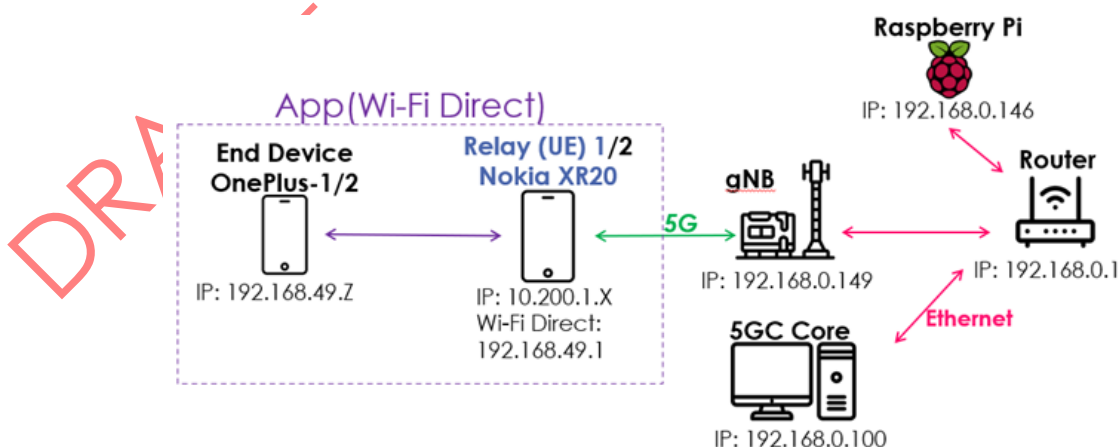


Figure 2-1: Overall architecture of the experimental setup.



## 2.3 Experimental methodology

Three main tests were performed in this work. First, the testing of the 5G network and its specifications, then the Wi-Fi Direct and its specifications and finally, the complete end-to-end communication. The parameters that were used to test the connections between the devices where:

- Round Trip Time (RTT): It is the duration in milliseconds (ms) that an ICMP Echo Request packet takes to travel from the Source to the Destination and return as an ICMP Echo Reply back to the Source. RTT is an important metric in determining the health of a connection on a network, to diagnose the speed and reliability of the network connections. The Echo Request packet is provided by the "ping" command, which will return the RTT of the sent Echo Request when its corresponding Echo Reply is received<sup>1</sup>. The ping was set to repeat the measurement 1000 times. Also, the packet size is the default size, which is 64 bytes<sup>2</sup>.
- Throughput: It is the traffic rate that a network channel can handle per unit time. Throughput depends on factors such as bandwidth, latency, payload size, packet size, network load, number of hops, and others. Throughput was measured using a standard tool called iperf3<sup>3</sup>.
- Jitter: It is the standard deviation between the transmitted signal delays. Its variation is usually due to network congestion, poor hardware performance and non-application of packet prioritization. The meaning has to do with the variation of a metric (e.g., delay) with respect to some reference metric (e.g., average delay or minimum delay). Jitter was also measured using iperf3.

Iperf3 was installed on the server and the other device (client) in order to exchange messages between them. The requested bandwidth was set to 200 Mbps, which is higher than what devices can support, so that it will allow the devices to reach their maximum speeds. In addition, the reporting intervals are set at 1 s, so every second, the system presents bandwidth, jitter and loss reports. Another parameter is the length of the test, which is set at 1000 packets. Moreover, for the throughput and jitter tests, it was decided to use User Datagram Protocol (UDP) to establish a low-latency and loss-tolerating connection. On the other hand, TCP protocol as a connection-oriented protocol, guarantees the reception of all packets. Therefore, TCP is safer and more reliable than UDP but it is slower and requires more resources. In conclusion, the UDP is better suited for applications that need fast and efficient transmission. The gNB and 5GC operate in a Linux operation where the libraries for these parameters were installed. On the phones, with Android as their operating system, the Android terminal emulator and Linux environment application were installed. Iperf3 is a terminal emulation and Linux environment application that works directly with no rooting or setup required.

The measurement applications used for these tests have been installed in the terminal of the devices. These applications are already installed for BS and 5GC, which run on Linux. In

---

<sup>1</sup> [https://thinksystem.lenovofiles.com/storage/help/index.jsp?topic=%2Fthinksystem\\_system\\_manager\\_11.50.1%2F649BE1AD-24F5-4B3D-892D-8AF14C37619C\\_.html](https://thinksystem.lenovofiles.com/storage/help/index.jsp?topic=%2Fthinksystem_system_manager_11.50.1%2F649BE1AD-24F5-4B3D-892D-8AF14C37619C_.html)

<sup>2</sup> N. V. Mnisi, O. J. Oyedapo and A. Kurien, "Active Throughput Estimation Using RTT of Differing ICMP Packet Sizes," 2008 Third International Conference on Broadband Communications, Information Technology & Biomedical Applications, Pretoria, South Africa, 2008, pp. 480-485, doi: 10.1109/BROADCOM.2008.76.

<sup>3</sup> <https://en.wikipedia.org/wiki/Iperf>

In addition, these commands have been installed via a terminal emulator application using the Termux interface on the android phones.

## 2.4 Experimental Results

### 2.4.1 5G network

The first set of tests focused on the 5G Network side between the 5GC/BS and the relay devices. The relay devices and the 5GC are connected through the gNB using 5G NR in SA mode. Two Nokia XR20 mobile phones were used as relays and placed near the BS station during the tests for higher performance. It is essential to mention that, at this point, these tests were not performed in an anechoic chamber. Therefore, the measurements performed could be affected by Electro-Magnetic Interference (EMI).

#### 2.4.1.1 Test between gNB and Relay

Although these tests and measurements were recorded between the gNB and the relays, in order for the packets to be authenticated, they should reach the 5GC, hence essentially the presented time here includes the time to go to the 5GC and back. Table 2-1 and Table 2-2 show RTT from gNB to Relay and reverse. RTT times, both download and upload, are adequately stable between the two relays. The average value of RTT is around 23.4 ms from gNB to relay, compared with RTT from relay to gNB, which is slightly higher, approximately 25.3 ms. Also, essential parameters represented in tables are standard deviation with maximum and minimum values. In detail, the standard deviation is around 6 ms, and the maximum value sometimes is over 80 ms, which shows that the measurements fluctuate, highlighting a significant network instability in terms of performance. Overall, measured RTT times are within the acceptable range for 5G SA eMBB slice according to 3GPP, Release 16 standards.

Table 2-1: RTT from gNB to Relay device.

gNB to Relay	Phone 1(ms)	Phone 2 (ms)
Average	23.45	23.41
Max	80.90	41.60
Min	12.00	10.00
Standard Dev	6.14	5.97

Table 2-2: RTT from Relay to gNB

gNB to Relay	Phone 1(ms)	Phone 2 (ms)
Average	25.16	25.31
Max	72.00	92.40
Min	10.20	9.33



Standard Dev	6.19	6.41
--------------	------	------

Table 2-3 and Table 2-4 present the results on throughput from gNB to the Relay device and reverse. Also, these tables present the reverse mode, which represents the upload speeds. The average value for download is around 110 Mbps. Also, the standard deviation for these measurements is 8.87 for Phone 1 and 4.76 for Phone 2. Moreover, the result for the Phone 1R and Phone 2R are 3.68 Mbps and 2.53 Mbps, respectively. Therefore, the test values portray that the upload speed is slower than download speed, as expected for this testbed. These results highlight again the erratic behaviour of User Equipment (UEs) in terms of stability and significant variation between devices.

Table 2-5 shows the connection from the Relay device to gNB. The difference between the previous tables is that reverse mode represents the download, and normal mode the download speed. The average value for Phone 1 and Phone 2 is 4.31 Mbps and 0.73 Mbps, respectively. The average values for download are 113 Mbps and 59.7 Mbps. However, the measurements in Table 2-4 between Phone 1 and Phone 2 are different, showing that the second phone has either a hardware issue or that its hardware is significantly different even though they are using the same network ICs. In addition, the same issue is represented in the reverse mode testing on the same phone, especially since the standard deviation value is around 21 Mbps.

Furthermore, the measurements between Table 2-3 and Table 2-4 are aligned. More specifically, the phones from Table 2-3 are almost the same as Phones in reverse mode since their measurements show the throughput from gNB to Relay device. Moreover, the measurements from Phone 1 & 2 R in Table 2-3 and Phone 1 & 2 in Table 2-4 are almost identical. In conclusion, these results and correlations are presented as expected.

Table 2-3: Measure the Throughput from gNB to Relay (R – Reverse mode)

gNB to Relay	Phone 1 R (Mbps)	Phone 2 R (Mbps)	Phone 1 (Mbps)	Phone 2 (Mbps)
Average	3.68	2.53	115.26	101.35
Max	6.47	3.92	142.00	119.20
Min	3.08	1.52	69.10	88.00
Standard Dev	0.36	0.34	8.87	4.76

Table 2-4: Measure the Throughput from Relay to gNB

Relay to gNB	Phone 1 R (Mbps)	Phone 2 R (Mbps)	Phone 1 (Mbps)	Phone 2 (Mbps)
Average	113.93	59.70	4.31	0.73

## D2.3 Enhanced IoT Underlying Technology (Final Version)

Max	136.80	111.20	7.49	5.68
Min	88.00	0.01	2.21	0.32
Standard Dev	7.98	21.96	0.66	0.23

At this test, we compared the network performance under different PLMNs and whether this affects the throughput between the BS and relay device. The Table XXX portrays the results from throughput between the Relay devices and gNB, in PLMN 999-99. This test is same with the previous test at Table 2-3 and Table 2-4, but the difference is only in PLMN, the previous test uses the PLMN 001-01, which is for the testing network. In conclusion, the measurements from Table 2-3, Table 2-4, and Table 2-5 are similar. Therefore, the selection for PLMN does not affect the network performance.

Table 2-5: Measure the Throughput from Relay to gNB with PLMN 999-99

PLMN: 999-99	gNB to Relay Phone 1 (Mbps)	gNB to Relay Phone 2 (Mbps)	Relay to gNB Phone 1 (Mbps)	Relay to gNB Phone 2 (Mbps)
Average	94.03	61.50	6.02	1.37
Max	108.80	92.00	13.68	2.96
Min	3.44	0.00	0.00	0.56
Standard Dev	7.98	21.96	0.66	0.36

Table 2-6 and Table 2-7 displays the results of the jitter. The results follow the same pattern with the throughput measurements in Table 2-3 and Table 2-4. An essential parameter on this test is the maximum value of the jitter, which sometimes reaches around 270 ms. This high value of jitter causes for some measurements to be delayed to be received, which shows that the network is highly affected by external parameters, such as EMI since all these experiments were not conducted in an anechoic chamber.

Generally, jitter values are within or close to 3GPP Release 16 standards in terms of averages however, the fluctuation of those values is much higher than the standards. Again, this can be attributed to the fact that the tests were not performed in an anechoic chamber.

Table 2-6: Measurements of Jitter between from 5GC Core to Relay

5GC Core to Relay	Phone 1 R (ms)	Phone 2 R (ms)	Phone 1 (ms)	Phone 2 (ms)
Average	5.53	8.05	0.15	0.19
Max	16.11	263.61	0.93	1.40
Min	2.02	3.86	0.05	0.07

Standard Dev	2.08	8.60	0.09	0.14
--------------	------	------	------	------

Table 2-7: Measurements of Jitter between from Relay to 5GC Core

Relay to 5GC Core	Phone 1 R (ms)	Phone 2 R (ms)	Phone 1 (ms)	Phone 2 (ms)
Average	0.30	0.86	4.78	25.92
Max	0.91	270.91	15.65	51.45
Min	0.02	0.07	2.07	10.77
Standard Dev	0.12	8.74	1.99	7.23

### 2.4.1.2 Test between 5GC Core and Relay device

This test focuses on the connection between the 5GC and Relay devices. It is almost the same test as the previous one because the connection between the Relay device and 5GC passes through the gNB. Therefore, this test aims to investigate the connection between 5GC and gNB. The system diagram in Figure 2-2 shows the measurement connections in yellow, while the connection between the devices is presented with green dashed lines. In summary, the results from the measurement expect to have the same results as the previous test.

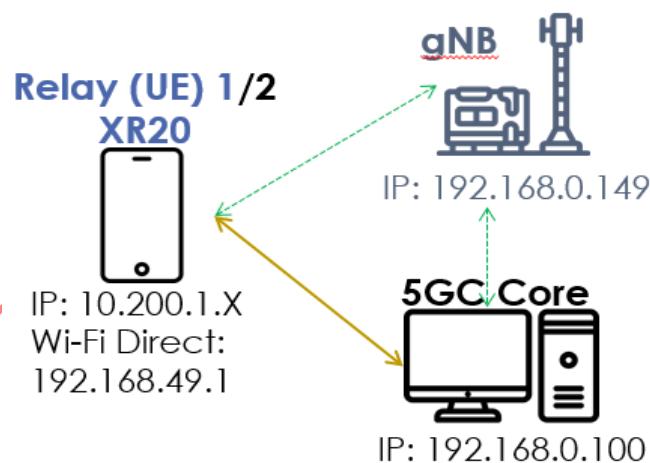


Figure 2-2: Test between 5GC Core and Relay device.

The first part of this test focuses on RTT. Table 2-8 presents the results from the 5GC Core to Relay devices, and Table 2-9 shows the results from Relay devices to 5GC. The results from gNB and relays are almost the same as those below. More specifically, the average value of RTT is around 23 ms in Table 2-8, and in Table 2-9, the average value is around 25 ms. In conclusion, the ethernet connection between the gNB and 5GC is shown not to affect the measurements.

Table 2-8: RTT from 5GC Core to Relay device

5GC Core to Relay	Phone 1 (ms)	Phone 2 (ms)
Average	23.23	23.21
Max	42.30	42.80
Min	12.70	12.40
Standard Dev	5.86	5.92

Table 2-9: RTT from Relay to 5GC Core

Relay to 5GC Core	Phone 1 (ms)	Phone 2 (ms)
Average	25.03	25.58
Max	44.4	44.40
Min	14.9	12.50
Standard Dev	5.91	5.95

Table 2-10 and Table 2-11 provide information for the throughput between Relay devices and 5GC. The measurements in Table 2-10 & Table 2-11 follow a similar pattern to Table 2-3 & Table 2-4, such as the download speed from Relay devices and also the upload speed fits the pattern.

Table 2-10: Measure the Throughput from 5GC Core to Relay (R – Reverse mode).

5GC to Relay b	Phone 1 R (Mbps)	Phone 2 R (Mbps)	Phone 1 (Mbps)	Phone 2 (Mbps)
Average	3.51	1.35	129.36	96.84
Max	8.04	1.52	144.00	108.00
Min	2.44	0.80	101.00	83.20
Standard Dev	0.51	0.14	11.30	4.30

Table 2-11: Measure the Throughput from Relay to 5GC.

Relay to 5GC a	Phone 1 R (Mbps)	Phone 2 R (Mbps)	Phone 1 (Mbps)	Phone 2 (Mbps)
Average	140.15	134.28	0.66	0.58

Max	143.00	135.20	1.49	5.04
Min	136.00	0.04	0.01	0.01
Standard Dev	18.97	1.01	0.30	0.20

Table 2-12 & Table 2-13 show the jitter between 5GC and Relay devices. The jitter values remain low, especially in download, but they are slightly higher than the results from the previous connection with gNB. Also, jitter value for the upload is higher than in Table 2-6 & Table 2-7, but they stay at the same level. The average value of the jitter from the Relay device to 5GC is around 6 ms for the Phone 1 and 14 ms for the Phone 2 in Reverse mode. These values can be compared with Table 2-12 and Test 1 and Test 2, in which the average values are around 26 ms and 32 ms. Also, in the opposite direction, the result is below 0.3 ms from 5GC to Relay device, in Table 2-13. On the other hand, the results in Table 2-13 are higher than the previous in Test 1, almost 4.5 ms, and in Test 2 is around 1.3 ms. This can be attributed to the capabilities of the 5GC to perform more tasks in parallel in addition to any external noise that could affect the measurements.

Table 2-12: Measure the Jitter between from 5GC Core to Relay

Relay to 5GC Core b	Phone 1 R (ms)	Phone 2 R (ms)	Phone 1 (ms)	Phone 2 (ms)
Average	5.79	14.47	0.16	0.30
Max	16.94	31.97	0.60	1.36
Min	2.28	7.36	0.04	0.06
Standard Dev	2.36	4.14	0.08	0.17

Table 2-13: Measure the Jitter between from Relay to 5GC Core

Relay to 5GC Core b	Phone 1 R (ms)	Phone 2 R (ms)	Phone 1 (ms)	Phone 2 (ms)
Average	4.29	1.32	26.80	32.29
Max	542.77	116.44	64.91	79.63
Min	0.04	0.04	10.25	14.02
Standard Dev	22.54	9.04	7.50	8.53

## 2.4.2 WiFi Direct Network

The second set of tests is related to the Wi-Fi Direct side of the network. In this test, two OnePlus8T mobile phones were used as end-devices, and two Nokia XR20 phones were used

as relays. The connection between the end-devices and the relays was achieved through the custom mobile/server application described in the previous section.

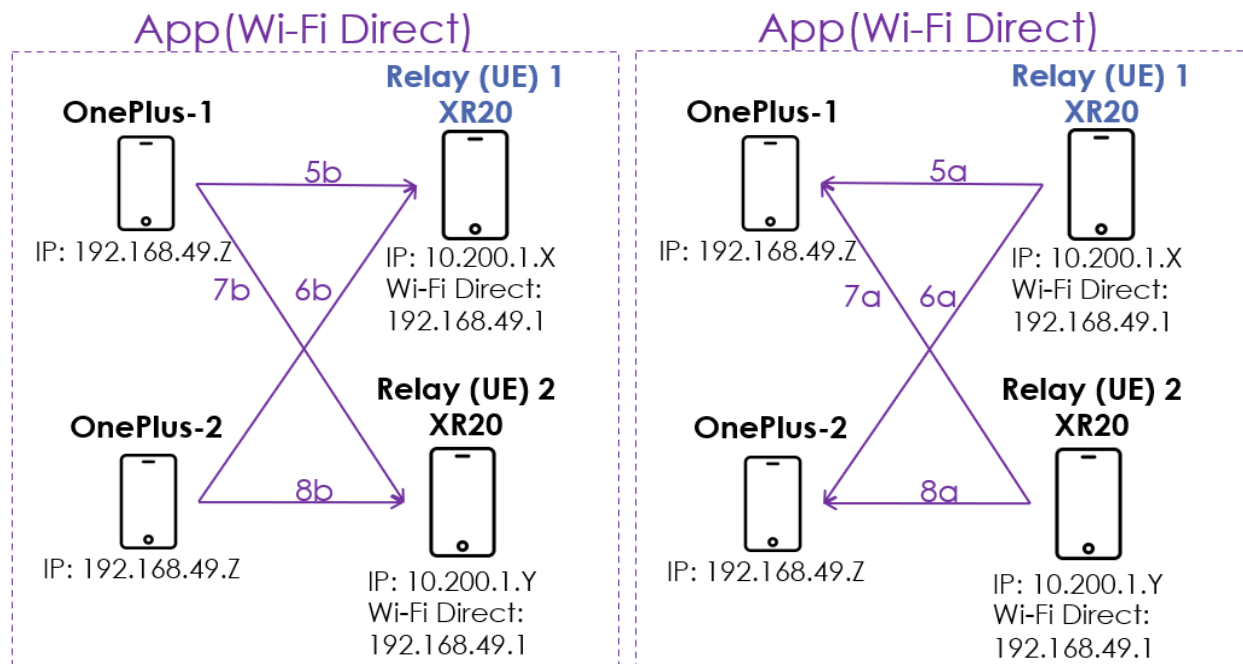


Figure 2-3: Testing of Wi-Fi Direct Network.

The first part of this test is a connection from the Relay device to the end-device as shown in Figure 2-3 and the results from the measurement are shown in Table 2-14. Firstly, all measurements are quite close to each other, which shows the system is repeatable and consistent. The average of these values is around 21 ms, also essential parameters on this table are the standard deviation, which is around 27 ms.

The second part of the RTT test is connection from End device to Relay device, which presents on the Figure 2-3. Table 2-15 presents the results of RTT in the opposite direction from the previous table. The average value of RTT from Table 2-15 is around 87ms, which is higher than the previous RTT, so the transmission from OnePlus is higher than from Nokia XR20. As a result, the RTT from the end-device is lower than from the relay device. In addition, the measurements in Table 2-15 follow the same pattern as those in Table 2-14 but with higher values. Also, the standard deviation is around 56 ms, which is high compared to the average value, also the difference between maximum and minimum value has increased. Therefore, these affect the phones or/and WiFi connection between the devices.

Table 2-14: Measure the RTT from End-Device to Relay

	5b (ms)	6b (ms)	7b (ms)	8b (ms)	Overall
Average	20.83	23.14	21.49	20.03	21.38
Max	276	388	346	268	388
Min	2.31	2.25	2.44	6.93	2.25

## D2.3 Enhanced IoT Underlying Technology (Final Version)

Standard Dev	27.59	30.38	31.59	18.37	26.99
--------------	-------	-------	-------	-------	-------

Table 2-15: Measure the RTT from Relay to End-Device

	5b (ms)	6b (ms)	7b (ms)	8b (ms)	Overall
Average	101.08	93.83	75.87	79.60	87.59
Max	523	422	406	413	523
Min	2.05	2.31	2.75	6.51	2.05
Standard Dev	65.82	57.29	51.43	51.44	56.50

In addition, Table 2-16 and Table 2-17 show the throughput between the devices which are connected with Wi-Fi Direct. The measurements show that the direction of testing is almost the same between the two directions. In continuation, the average value of the throughput for all measurements is around 47 MB/s, except for the Test 5a that the connection between the end-device and the relay faced a significant issue. In conclusion, these values correlate with the standard of Wi-Fi Direct. The results from Table 2-17 present some fluctuations, especially the connections with a specific end-device. In detail, the first end-device reaches high values of throughput, which is around 65 MB/s. On the other hand, the second end-device presents lower performance having an average value is around 33 MB/s.

Table 2-16: Measure the Throughput from End-Device to Relay

	5a (MB/s)	6a (MB/s)	7a (MB/s)	8a (MB/s)	Overall
Average	25.23	48.41	45.55	46.95	41.54
Max	51.90	56.90	57.00	56.80	57.00
Min	5.07	8.07	7.00	9.91	5.07
Standard Dev	13.59	11.24	11.60	15.86	13.07

Table 2-17: Measure the Throughput from Relay to End-Device

	5b (MB/s)	6b (MB/s)	7b (MB/s)	8b (MB/s)	Overall
Average	68.55	33.29	60.97	32.89	48.93
Max	81.2	40	80.4	41.5	81.20
Min	15.2	6.36	19.9	6.63	6.36

Standard Dev	16.09	7.11	16.14	7.94	11.82
--------------	-------	------	-------	------	-------

It is clear from the measurements that this side of the network experiences much higher RTT times, approximately  $87.6 \pm 13.49$  ms in the download mode. On the other hand, in the upload mode, the RTT times are approximately  $21.37 \pm 1.77$  ms. Hence, upload RTT times are similar to those of the 5G network, but the download side experiences approximately triple RTT times. It is also interesting that for the download RTT, the relay plays the most significant role since both measurements using relay number 1 show higher RTT times. In the upload mode, RTT times are significantly stable. In terms of throughput, it is obvious that in the download mode, the WiFi Direct side of the network shows speeds of approximately  $48.93 \pm 19.63$  Mbps, which is significantly slower than 5G. It is important to state that in the download mode, the throughput is mostly defined by the end-device and not the relay since both experiments with end-device number 2 show much slower throughput. In this case, end-devices have 2 different Android versions (End-device 1 Build: KB2003 11 C.33, Baseband ver:Q V1 P14, Kernel:4.19.157-perf+; End-Device 2 Build: KB2003 11 C.20 Baseband ver:Q V1 P14, Kernel:4.19.157-perf+), which can significantly affect the network performance. In terms of the upload mode, speeds of  $41.54 \pm 16.31$  Mbps have been recorded showing much higher throughput than 5G.

### 2.4.3 End-to-End D2D Communication Testing

In this last set of tests, the end-to-end setup of this D2D communication was demonstrated. The end-device and the relay connect through Wi-Fi Direct using the custom mobile/server application. The relay is also connected to the 5G BS using 5G NR. Initially, device and relay discovery are achieved through the mobile/server application and secondly message exchange from the server (Raspberry Pi) to the end-device is demonstrated using the configuration shown in Figure 2-4. Also, the server presents all connections between the phones and links through the connection, the alert message sends to phones. In conclusion, the target of this test is to establish this connection and calculate the transmission time.

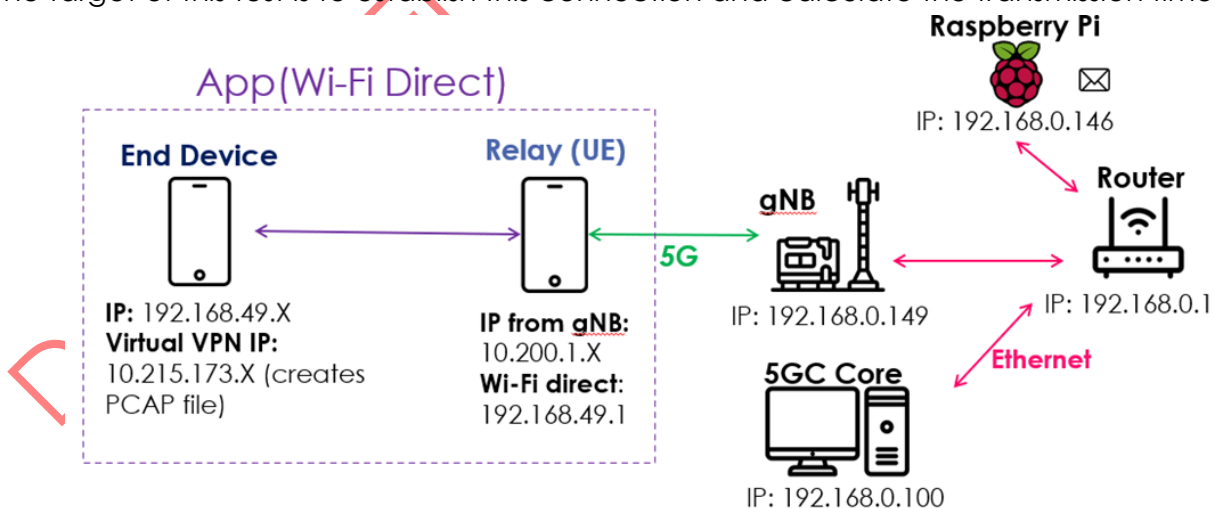


Figure 2-4: End-to-End experimental setup.

We analysed the reception times from a pcap file to calculate the transmission time. More specifically, on the raspberry pi where the server is hosted, we installed tcpdump, which sniffs



the network traffic of the channel where the 5GL is located. In the case of the phones, a pcap capture application was installed, which creates a virtual vpn on the phone and then collects the network traffic from the phone. As a result, the pcap file on the server stores the time the message was sent, while the pcap file on the android phones stores the time instant of reception. Therefore, the transmission time is the difference between the time transmitted from the Raspberry Pi and the time received from the end device. Figure 2-5 presents the results of 15 different measures for transmission time from Raspberry pi to end device. The overall transmission time needed to send the message from the server to the end-device has been measured to be approximately 91 ms with a 28 ms standard deviation. Adding the RTT times for the individual segments of travel, the total time is calculated to be approximately 112 ms. Acknowledging that RTT times involve the bi-directional travel of the packet, it is expected that this method would give higher times than Figure 3-5. End-to-End Testing setup the real ones. However, the calculation and measurements are significantly close to each other, confirming the correct order of magnitude in transmission time.

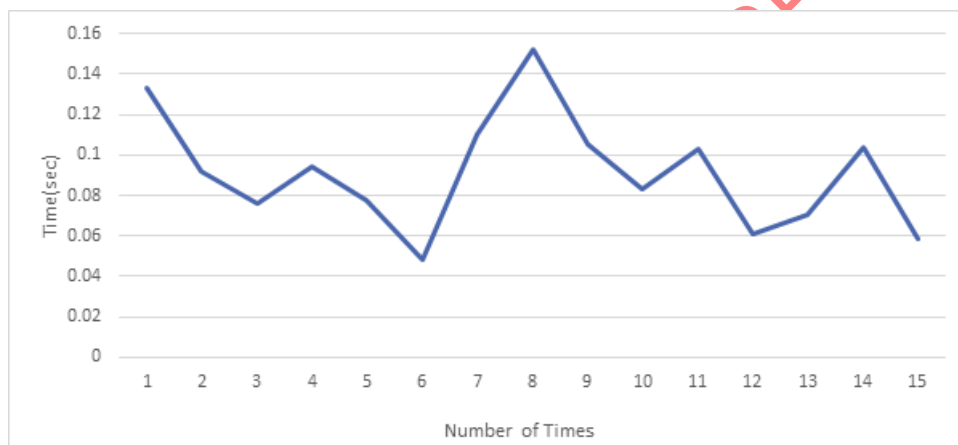


Figure 2-5: RTT times versus number of measurements for End-to-End tests.

## 2.4.4 Relay selection

Sorbonne University's android application is also responsible for selecting the best relay device during the coverage extension procedure. Figure 2-6 shows the diagram of this test where the identification number of each device is located under it through the ID tag.

As soon as the relay selection procedure begins, the relays maintain communication with all possible End-Devices in their coverage range. During this instant, the End-Devices provide their metrics to the relay devices for 1 min. The goal is to provide the relays with the necessary information to evaluate the condition of each End-Device. Then, the relays directly connect to the relay they selected, notifying their selection to the server located in the raspberry pi.

The next test was to repeat the previous test, but this time the relay device that had lower percentage of battery is charged. As mentioned previously, the percentage of battery is at this stage the main factor for the relay selection. In conclusion, the test presents the end devices select the best relay device that has the highest percentage of battery.

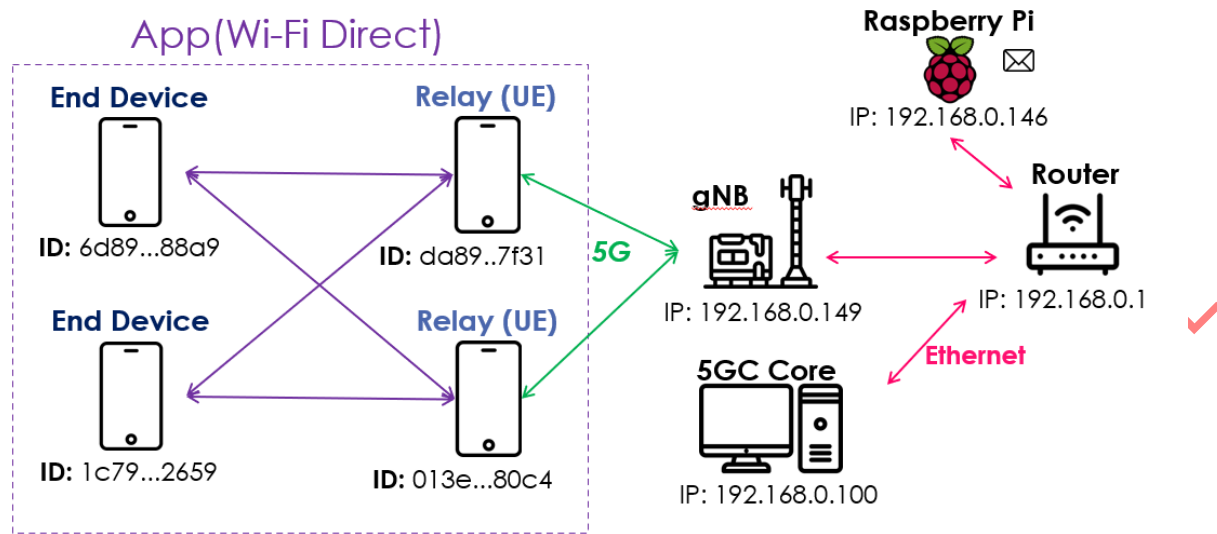


Figure 2-6: Architecture for relay selection.

## 2.5 Conclusions

D2D communication is a practical solution that will be more extensively used in the near future by exploiting technologies such as Wi-Fi or cellular extensions. In IoT-NGIN, we explored and researched solutions based on D2D, mostly to address the important topic of coverage extension that fits the case of IoT particularly. We first developed the methodology, the tool and the framework to design a practical solution. Then, we did an extensive D2D link characterization to assess the link's characteristics and the criteria for the relay selection. Moreover, we integrated this solution into a private 5G LAN that is also part of the IoT-NGIN design. Finally, we demonstrated and evaluated this setup as a function of various key parameters. Our work has been published (one conference publication and one submission being evaluated). The tools are available as open-source software).

### 3 Enabling 5G to support protocols for deterministic communications

The usage of 5G for private industrial networks needs to support deterministic communications which have been designed originally to optimize time sensitive (i.e., TSN) communications between machines. The first requirement is to synchronize all the devices (i.e., wired and wireless) that have to communicate following pre-defined time slots to ensure resilience. Thus, TSN requires synchronising mobile network UEs and devices connected to fixed LAN.

The industrial LAN may also consist of TSN-enabled Ethernet bridges. The latest release of 5G specification supports the fully centralised TSN configuration model, where a central controller should be able to configure both Ethernet and 5GS bridges as a unified network. The 5GS supports the whole industrial network, both Medium Access Control (MAC) learning and flooding based forwarding as well as the static forwarding configured by the central controller need to be supported. 3GPP has defined that a 5GS can be modelled as one or more virtual TSN bridges.

The CNC is the entity in the TSN network that has complete knowledge of the network topology and is responsible for configuring the bridges to enable transmission of TSN streams from source to destination. The 5G control plane is interacting with CNC via the TSN Application Function (AF) which maps between the TSN parameters and the 5GS parameters. The TSN AF reports the 5GS bridge capabilities such as minimum and maximum delays between every port pair and per traffic class, including the residence time within the UE and DS-TT via TSN-AF to CNC.

Topology discovery information based on the widely adopted standard IEEE 802.1AB Link Layer Discovery Protocol (LLDP) is also exposed. The TSN AF also exposes its TSN capabilities like the support for scheduled traffic and per-stream filtering and policing (PSFP) as specified in IEEE 802.1Q-2018, in case that they are supported by all of the ports. The CNC obtains the 5GS bridge VLAN configuration from TSN AF according to IEEE Std 802.1Q.

The TSN AF shall be pre-configured (e.g. via OAM) with a mapping table. The mapping table contains TSN traffic classes, pre-configured bridge delays (i.e. the preconfigured delay between UE and UPF/NW-TT) and priority levels. The CNC reads the capabilities of all bridges and calculates the traffic paths and schedules in the network. The CNC then provides the bridge configuration to the 5GS through the TSN AF, which contains, e.g., scheduled traffic, PSFP, and traffic forwarding information. In order to support QoS for Ethernet and TSN traffic, the traffic flows are mapped to 5G QoS flows. The CNC configures the traffic handling in the 5GS bridge for the different traffic classes according to the capabilities that have previously been reported by the 5GS bridge. The 5GS maps the Ethernet/TSN traffic classes or TSN traffic streams to the corresponding 5G QoS flows.

3GPP has defined 5G VN groups consisting of a set of UEs using private communication for 5G-LAN type services. A 5G VN group can be utilized for IP or Ethernet based services. A specific data network, identified by a data network name (DNN), is one of the possibilities to realise a 5G VN group, where the VN group can be either provided by Operation and Management (O&M) or by an TSN-AF. 5G VN group where the SMF has full control of the Ethernet network topology among the 5G VN group members (by control of forwarding decisions on all Ethernet PDU sessions from different UEs).

For a centrally managed Ethernet network, it is required that the NMS/CNC can configure the VLAN handling for all bridges and all ports, including the 5GS bridge, as specified in IEEE 802.1Q.

### 3.1 Introduction

CMC has been focusing in the design of 5G core specifically for Non Public Networks (NPN) primarily industrial networks that require deterministic communications following TSN features. Therefore, CMC 5G core includes the latest 3GPP standard specifications including all the necessary network functions (NF) for connecting UE devices to fixed LAN and become native TSN devices.

UE is defined to have attached functions of time stamping in data frames, using the device side of the TSN translator (DS-TT). On this Network Function, a step of time synchronisation is implemented using the TS<sub>i</sub> from the Suffix field of the gPTP messages (Sync or Follow Up messages), as it has been defined on 24.535 from 3GPP.

In order to achieve this function a 5G-Modem is integrated in a NPN following an inherent structure based on a 5G component, a TSN packet handler and wired network components. These last two components may be integrated in the same hardware such as a microprocessor but following the TSN standards defined on IEEE 802.1 Qbv, 802.1 CB, 802.1 As and 802.1 Qbu.

In order to demonstrate the TSN functionality CMC has deployed the 5G core in ABB living labs following the topology below. In this setup we demonstrate successful time synchronization of TSN devices connected to UE device with fixed TSN devices connected to LAN.

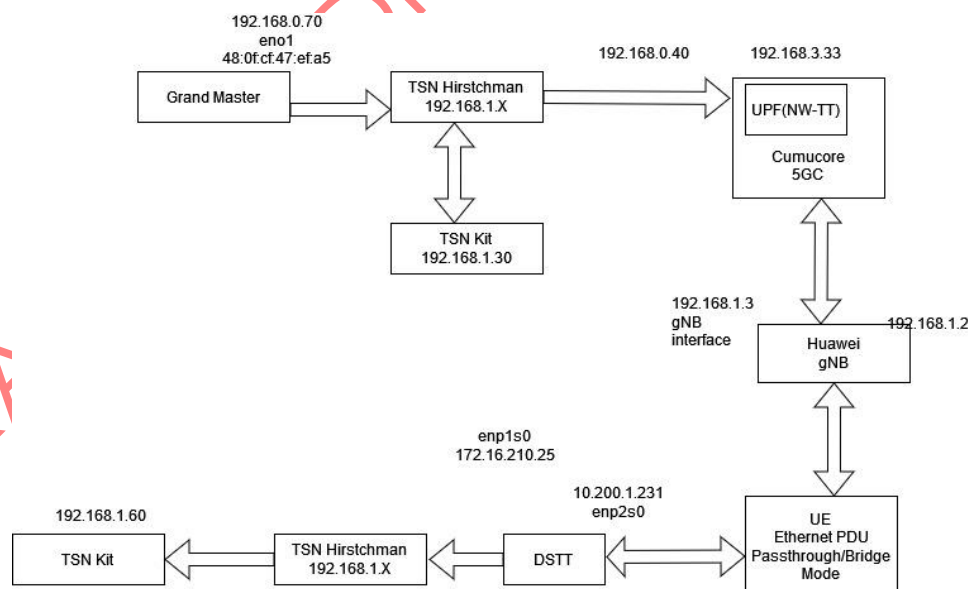


Figure 3-1: TSN network deployed at ABB living labs for time synchronization testing.

## 3.2 Results

After running the time synchronization process to get the UE in sync with the fixed devices we obtained following results

Table 3-1: Measure the time synchronization in TSN device connected to UE

Location	Results
UE Local Time	1677594032.510012643
Grand Master Time	1677594032.495004032
Offset	-15008611

Table 3-2: Measure the time synchronization in fixed TSN device

Location	Results
UE Local Time	1677594273.198258969
Grand Master Time	1677594273.198259137
Offset	168

The system was installed in ABB living labs as shown in the following figure for additional measurements

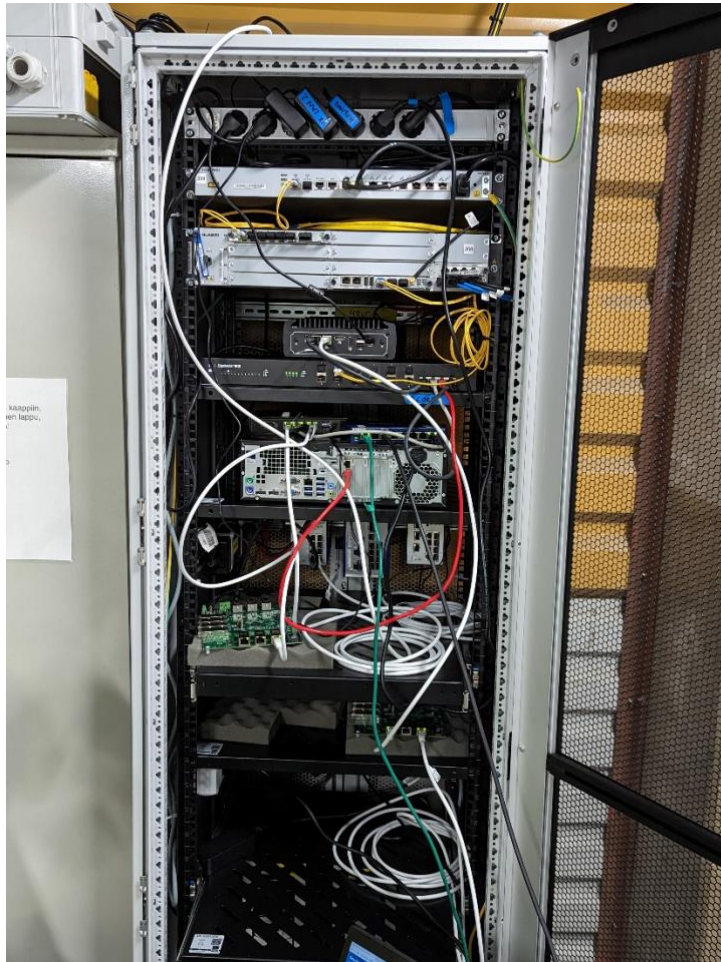


Figure 3-2: Equipment deployed at ABB living labs for time synchronization testing.

### 3.3 Conclusions

The testing of TSN network was conducted both in CMC laboratory and in ABB living lab premises. The setup uses unique system that provides Ethernet PDU connection over 5G network. The commercial 5G networks only support IP PDU sessions for data exchange between mobile devices and data networks such as public Internet. Instead, Non-Public Networks (NPN) for industry requires Ethernet PDU together with 5GLAN functionality. This setup includes the support for Ethernet PDU which is uniquely available currently in CMC 5G Core. The Ethernet PDU allows to transfer gPTP messages from Grand Master (GM) located in fixed LAN to the 5G modem which will send the gPTP messages to DS-TT for synchronizing the moving devices.

The preliminary results are shown in Table 3-1 and Table 3-2, where the mobile TSN device is synchronized but with different offset compared to the synchronization achieved by the fixed TSN device connected directly to the GM in the fixed LAN. The deployment in ABB labs had few limitations that were blocking the gPTP messages to reach the mobile TSN device. The end result is that gPTP was running over the 5G network using the Ethernet PDU but the base station disconnected the mobile and synchronization was lost.



This behaviour of the base station caused that gPTP messages did not reach the mobile TSN and went out of synch. Therefore, for providing reliable time synchronization the Ethernet PDU session should be supported not only in the 5G Core but also in the base station and 5G routers. Moreover, the jitter of the delay in the 5G radio link needs to be minimized to ensure the offset of the clocks in both mobile and fixed TSN are aligned.

Thus, initial results show the gPTP can be transferred over the 5G radio to certain degree of accuracy to synchronize mobile devices but still requires improvements to support natively Ethernet PDU and low delay jitter for reaching high levels of synchronization accuracy.

DRAFT - PENDING EC APPROVAL

## 4 Enhancing 5G ease of use through improved 5G APIs

In this chapter, we present the final version of the 5G-resource-management API, which is the major outcome of the work from task 2.3. The chapter is split into an introductory section followed by the latest status of the API and results of tests conducted with the API.

### 4.1 Introduction

In D2.2 *Enhancing IoT Underlying Technology*, the first revision of the IoT-NGIN 5G API was presented. The main goal of this API is to increase the ease of use for application developers and decrease the required knowledge of the underlying infrastructure. This is done by utilizing a three-layer approach. As shown in Figure 4-1 the top layer is either the field devices or the cloud or edge cloud application. On the middle layer the API Server available to the end user and an infrastructure specific adapter is placed. Finally, on the bottom layer, the interfaces to the specific infrastructure are connected. For more information about this approach and information about the planned API functionality deliverables D2.1 and D2.2 can be consulted.

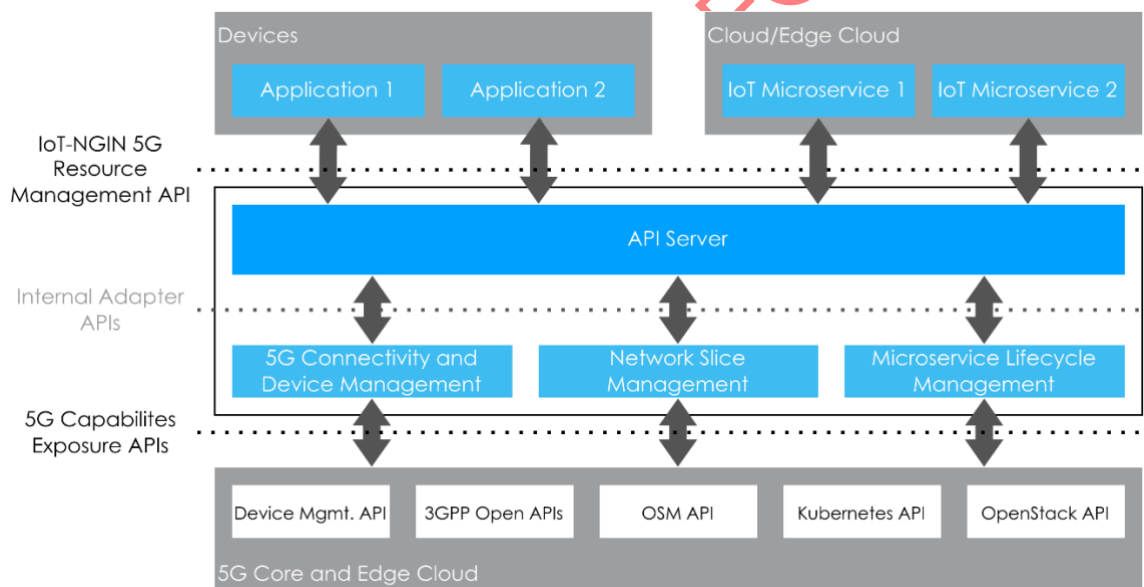


Figure 4-1 IoT-NGIN 5G resource management API

### 4.2 Final API

The latest version of the API that is tested in a simulated environment and is described as an Open API Specification found on the IoT-NGIN's project Gitlab<sup>4</sup>. This chapter will list the functionalities and describe the use cases of the different features. An updated list is

<sup>4</sup> [https://gitlab.com/h2020-iot-ngin/enhancing\\_5g\\_api](https://gitlab.com/h2020-iot-ngin/enhancing_5g_api)



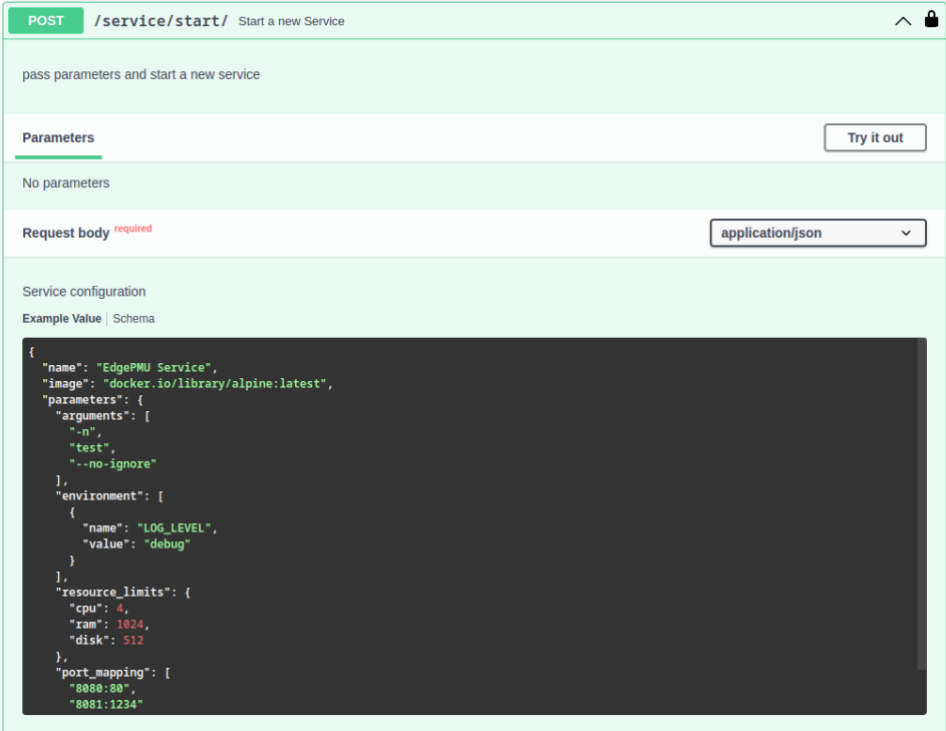
presented, compared to D2.2. As already described in D2.2 the API features are grouped into three different categories. The first category (5G Connectivity and Device Management) controls the devices and allows combined actions on groups of devices. The second category (Microservice Lifecycle Management) provides features for the control of services running on the cloud or edge cloud infrastructure. Finally, the third category (Network Slice Management) is controlling the connectivity between the devices and the services running on top of isolated network slices.

Table 4-1: API calls

Operation	Short description
<b>5G Connectivity and Device Management</b>	
/device	Register device with API.
/device/{device-id}	Query device information or unregister device.
/device/{device-id}/qos	Query device quality-of-service parameters or change QoS Parameters for Device.
/device/{device-id}/qos/subscribe	Subscribe to changes in the quality of service of a device to trigger an event accordingly
/device/{device-id}/location	Query device location.
/device/{device-id}/location/subscribe	Subscribe to changes in the location of a device to trigger events
/group	Create a new device group
/group/{group-id}	Add or remove device to or from group and query for the group members
<b>Microservice Lifecycle Management</b>	
/service/start/	Start a new service.
/service/{service-id}	Get Service information or terminate a service.
/service/{service-id}/status	Get the status of a service
/service/{service-id}/resources	Get or update the resources that a service is using on that are available for the service
/service/{service-id}/migrate	Migrate a service to a different target node
<b>Network Slice Management</b>	
/slice	Create a new network slice
/slice/{network-id}	Get the information of a slice or remove the network slice

<code>/slice/{network-id}/devices</code>	Add or remove a device from the network slice or list the devices that a currently part of the network slice
<code>/slice/{network-id}/qos/subscribe</code>	Subscribe to changes in the quality of service for the network slice
<code>/slice/{network-id}/services</code>	List all services that running on a specific network slice

Explaining all calls would go beyond the scope of this document. For illustration purposes, we explain the `POST /service/start/` call more in detail.



**POST** `/service/start/` Start a new Service

pass parameters and start a new service

**Parameters** Try it out

No parameters

**Request body** required application/json

**Service configuration**

Example Value | Schema

```
{
  "name": "EdgePMU Service",
  "image": "docker.io/library/alpine:latest",
  "parameters": {
    "arguments": [
      "-n",
      "test",
      "--no-ignore"
    ],
    "environment": [
      {
        "name": "LOG_LEVEL",
        "value": "debug"
      }
    ],
    "resource_limits": {
      "cpu": 4,
      "ram": 1024,
      "disk": 512
    },
    "port_mapping": [
      "8080:80",
      "8081:1234"
    ]
  }
}
```

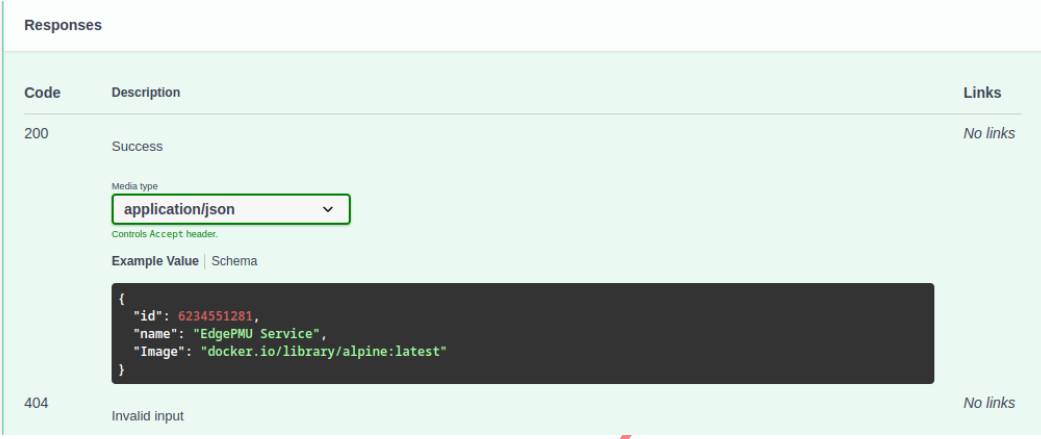
Figure 4-2: Open API UI

This call starts a new service on the (Edge-)Cloud backend. Therefore, it is necessary to pass the respective parameters for that service in the call's body. These parameters can be seen in Figure XY, which shows the rendered API.

Every service needs to have a human-readable name for distinction and maintenance. As we build upon container technologies, a docker image in standard notation is also required. In theory, this would suffice, but such a minimal service would barely be of any use. Additional parameters can be passed to the API to configure the running images. For once, there are the arguments and environment variables, which are passed into the container to configure the running software. In the API example we have several service specific command-line parameters like `--no-ignore` and environment variables like `LOG_LEVEL` to set the log verbosity. It should be noted that the environment variables are key-value pairs.

The next parameter is `resource_limits`. As the name already implies, these can be used to set quotas on the new service. This feature can be useful, for instance, to reduce costs for the service or ensure sufficient remaining resources for other services to be run on that network slice.

The last but probably most important parameter are the `port_mappings`. These expose the services' container ports, which is crucial for any cloud service. In the example, the container's port 80 and 1234 are mapped to the host's ports 8080 and 8081 respectively.



Code	Description	Links
200	Success	No links
Media type: <input type="text" value="application/json"/> Controls Accept header: Example Value   Schema <pre>{   "id": 6234551281,   "name": "EdgePMU Service",   "Image": "docker.io/library/alpine:latest" }</pre>		
404	Invalid input	No links

Figure 4-3: OpenAPI visualization of the `get_service` call's response.

A response to such a call is depicted in Figure 4-4. It contains the service's name and image, which in this case are the same as what was passed to the call in the first place. However, as the same schema is used in multiple places in the API, this information is still relevant. The second return element is the "id" value of the response. This is a 64-Bit identifier, which can later be used to select that specific service in other calls, such as `DELETE /service/{service_id}`.



```
> curl -X 'POST' \
  'http://127.0.0.1:8001/api/v1/service/start/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "Helloworld",
    "image": "nginxdemos/hello",
    "parameters": {
      "resource_limits": {
        "cpu": 2,
        "ram": 1024
      },
      "port_mapping": [
        "8080:80"
      ]
    }
  }'
{"id":"899942548673233147","name":"helloworld","Image":"nginxdemos/hello"}
```

Figure 4-4: Call and response of `start_service` in the command line.

Figure 4-4 shows a call to that API using the Linux command line tool "curl", which executes http requests. In the example a "hello world" server is started, which can then be accessed at port 80 of the edge cloud infrastructure.

## 4.3 Implementation

The core component is the API server. Additionally, we implemented three different adapters and tested either against a lab infrastructure or a simulated infrastructure. More setup specific adapters can be added easily. This showcases the versatility of the API and adapter approach for different types of infrastructure.

### 4.3.1 IoT-NGIN API Server

The API Server provides the interface to the user application. This server is implemented using the Rust programming language, which is chosen for its reliability and security, resulting from strong typing and extensive runtime checks. The code of the server can be found in the projects Gitlab repository<sup>5</sup>.

The API Server is developed as a stateless server that interfaces to the different API Adapters. It is built using the OpenAPI tools<sup>6</sup> which ensure that all calls are actually handled, that datatypes are correct, and the responses are in line with the specified API. In the backend, the highly performant hyper<sup>7</sup> library is used, which is among the most performant and thus also efficient frameworks for http communication.

The API Server exposes all the above-mentioned API calls and can easily be extended in the future by updating the open API configuration and the implementation of the modified or added API function calls.

### 4.3.2 Unikernel Adapter

The Unikernel Adapter can be found on Gitlab.com<sup>8</sup>. This adapter enables the API to start and stop services that are running a Unikernel application. The Adapter provides a rudimentary API interface for Kubernetes and specifically the Unikernel application. This adapter also stores the current state of the service and can in the future be extended to store more state information about the service that is instantiated. The goal is to abstract the adapter specific state information away from the customer. This allows for a much simpler API and higher flexibility in the implementation of different adapters.

Table 4-2: Unikernel Adapter API calls

Operation	Short description
/service	Create and start a service
/service/{id}	Get status or delete a specific service

<sup>5</sup> [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/5g-api-server](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/5g-api-server)

<sup>6</sup> <https://github.com/OpenAPITools/openapi-generator>

<sup>7</sup> <https://hyper.rs/>

<sup>8</sup> [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/5g-api-adapters/endpoint-unikernel](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/5g-api-adapters/endpoint-unikernel)

### 4.3.3 Slice Management Adapter

The Slice Management Adapter provides an interface to the vendor specific infrastructure of the I2CAT slice manager. This adapter implements a rudimentary interface to instantiate a service and enable connectivity to the edge cloud infrastructure, where i2CAT's Slice Management API is deployed. In the latest version, the following API calls are implemented. The currently for testing purposes implemented `service_start` call actually executed four API requests as depicted in Table 4-3. The adapter can be found in the project's Gitlab repository<sup>9</sup>.

Table 4-3: I2CAT Backend API calls.

Operation	Short description
<code>/service_start</code>	This adapter capability implements the start service calls

### 4.3.4 Slice Management simulator

For testing purposes, a simulated backend for the adapter was developed. This simulator is based on the OpenAPI definition for the I2CAT slice manager. It implements a minimal subset to allow for testing the calls ran by the nsm-adapter. The simulated backend supports the following calls including their specific parameters. The simulator stores the state of the system, thus allowing to read the current state as well as create new instances as described in Table 4-4. The simulator can also be found in the project's Gitlab repository<sup>10</sup>.

Table 4-4: Slice Management Simulator calls.

Operation	Short description
<code>/compute_chunk</code>	This call creates a new compute chunk and returns the parameters of the chunk
<code>/slice</code>	This call creates a new slice
<code>/network_service</code>	This call creates a network service
<code>/network_service_instance</code>	This call starts an instance of the above created network service

<sup>9</sup> [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/endpoint-nsm](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/endpoint-nsm)

<sup>10</sup> [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/5g-api-adapters/endpoint-simulator](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/5g-api-adapters/endpoint-simulator)

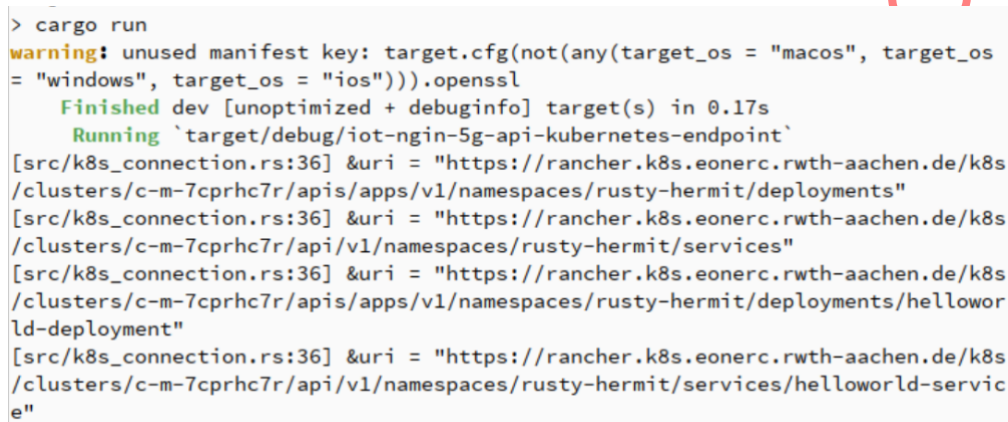
## 4.4 Results

In this section two different demonstrations of the API server are described.

For all the test the API Server as well as the corresponding adapters were running and the calls to the API Server were produced via the Open API Web UI or a curl command.

### 4.4.1 API Server

The API server was tested in a qualitative manner. As the code generation ensures that stubs for all specified functions are generated and the forwarding to the respective adapter is generalized, the functionality of all calls could be verified easily.



```
> cargo run
warning: unused manifest key: target.cfg(not(any(target_os = "macos", target_os = "windows", target_os = "ios")))
openssl
Finished dev [unoptimized + debuginfo] target(s) in 0.17s
Running `target/debug/iot-ngin-5g-api-kubernetes-endpoint`
[src/k8s_connection.rs:36] &uri = "https://rancher.k8s.eonerc.rwth-aachen.de/k8s/clusters/c-m-7cprhc7r/apis/apps/v1/namespaces/rusty-hermit/deployments"
[src/k8s_connection.rs:36] &uri = "https://rancher.k8s.eonerc.rwth-aachen.de/k8s/clusters/c-m-7cprhc7r/api/v1/namespaces/rusty-hermit/services"
[src/k8s_connection.rs:36] &uri = "https://rancher.k8s.eonerc.rwth-aachen.de/k8s/clusters/c-m-7cprhc7r/apis/apps/v1/namespaces/rusty-hermit/deployments/helloworld-deployment"
[src/k8s_connection.rs:36] &uri = "https://rancher.k8s.eonerc.rwth-aachen.de/k8s/clusters/c-m-7cprhc7r/api/v1/namespaces/rusty-hermit/services/helloworld-service"
```

Figure 4-5: The IoT-NGIN 5G Resource Management API server in action.

### 4.4.2 Unikernel Demo

In this section, a demo of the Unikernel API adapter is described. The initial call is sent to the IoT NGIN API Server and then forwarded to the Unikernel adapter. That in result calls the Kubernetes API and starts the selected service. Currently two examples are available. The first example starts a NGINX that shows the users IP address and the second one is a hello world Unikernel application that is serving a static html website. It is to be noted that the NGINX example uses the standard container infrastructure and the Unikernel example starts a Unikernel using runh. A screenshot of the Unikernel application can be seen in Figure 4-6. The Rancher user interface of Kubernetes, showing the running Unikernel application is shown in Figure 4-7.



Figure 4-6: Unikernel application started with IoT NGIN 5G API.

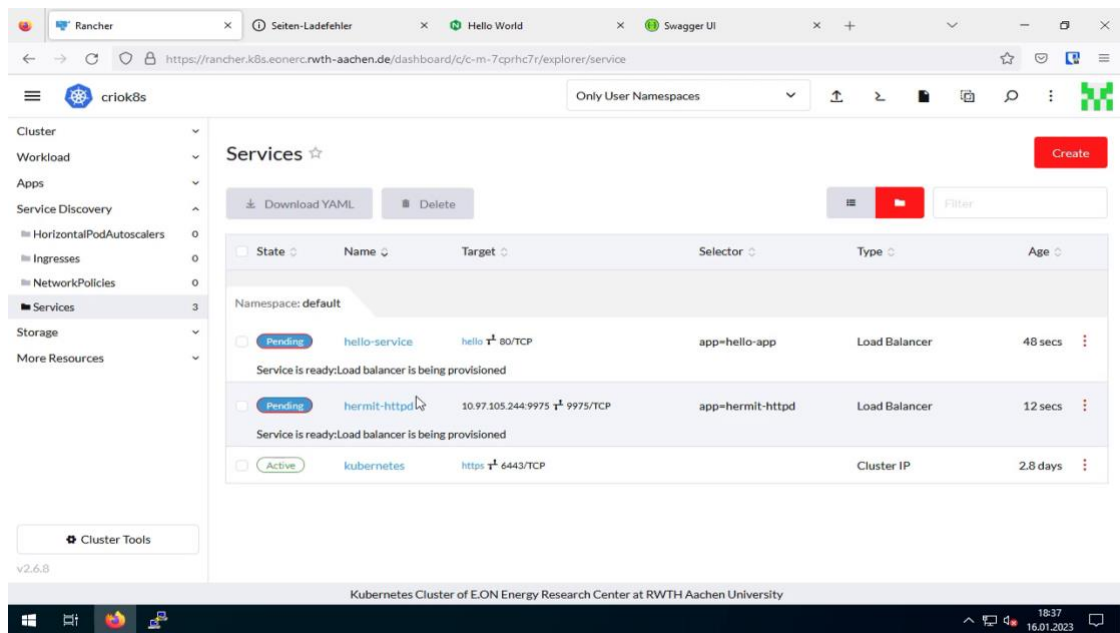


Figure 4-7: Rancher UI after starting a Unikernel hello world service.

### 4.4.3 Slice Manager Demo

In this test the resource management API is called, to start a service. This then results in four subsequent calls to the slice manager simulator. The API simulator is depicted in Figure 4-8. Figure 4-8 there it can be seen that the four calls for chunk, slice, network service creation and network service instantiation is received. These calls are executed by the API adapter who got a `start_service` call.

```
source /home/mrx/devel/iot-ngin/example-adapter/vEnv/bin/activate
~/devel/iot-ngin source /home/mrx/devel/iot-ngin/example-adapter/vEnv/bin/activate
~/devel/iot-ngin /usr/bin/env /home/mrx/devel/iot-ngin/example-adapter/vEnv/bin/python /home/mrx/.vscode/extensions/ms-python/python-2022.16.1/pythonFiles/lib/python/debugpy/adapter/../../debugpy/launcher 37463 -- /home/mrx/devel/iot-ngin/endpoint-simulator/simulation.py
* Serving Flask app 'simulation'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:8081
* Running on http://134.61.66.38:8081
Press CTRL+C to quit
Create chunk for EdgePMU Service
127.0.0.1 - - [30/Mar/2023 14:37:51] "POST /api/v1.0/compute_chunk HTTP/1.1" 200 -
Create slice for user 5b63089158f568073093f70d
127.0.0.1 - - [30/Mar/2023 14:37:51] "POST /api/v1.0/slic3 HTTP/1.1" 200 -
Create network service captive_portal
127.0.0.1 - - [30/Mar/2023 14:37:51] "POST /api/v1.0/network_service HTTP/1.1" 200 -
Start instance of ns EdgePMU Service
127.0.0.1 - - [30/Mar/2023 14:37:55] "POST /api/v1.0/network_service_instance HTTP/1.1" 200 -
```

Figure 4-8: Slice Manager command line tool.

The curl version for the call to the API Adapter is depicted in Figure 4-9 and the result of the above shown calls is shown in Figure 4-10.

```
curl -X 'POST' \
  'http://127.0.0.1:8080/api/v1/start' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "image": "docker.io/library/alpine:latest",
    "name": "EdgePMU Service",
    "parameters": {
      "arguments": [
        "-n",
        "test",
        "--no-ignore"
      ],
      "resource_limits": {
        "cpu": 4,
        "disk": 512,
        "ram": 1024
      }
    }
  }'
```

Figure 4-9: curl command to start a service.

Request URL	
http://127.0.0.1:8080/api/v1/start	
Server response	
Code	Details
200	<p>Response body</p> <pre>{   "Configuration": [     "tcp/8085:80",     "villas-node /config.conf"   ],   "Image": "edgePMU",   "Limits": "1000",   "id": 534,   "name": "EdgePMU Service" }</pre> <p>Response headers</p> <pre>access-control-allow-origin: http://127.0.0.1:8080 connection: close content-length: 163 content-type: application/json date: Thu, 30 Mar 2023 12:47:14 GMT server: Werkzeug/2.2.3 Python/3.10.9 vary: Origin</pre> <p>Responses</p>

Figure 4-10: OpenAPI visualization of the start\_service call.

## 4.5 Conclusions

In this chapter a simplified API is proposed that allows more software developers to create apps that can utilize the new functionalities provided by the 5G network. This in turn allows for a better adoption and the creation of new use cases in the future. The basic functionalities are shown with two different examples.



## 5 Secure Edge Cloud Framework for micro-services

In this chapter the work related to the development of a framework to allow more secure execution of micro services is described. This chapter presents the resulting framework.

### 5.1 Introduction

As described in deliverable 2.2, typical cloud and edge cloud infrastructure is based on containers, which allows the existence of multiple isolated user spaces. Containers are based on OS-level virtualization, where the applications are bundled in logical namespaces and the kernel ensures isolation between these namespaces. The left side of Figure 5-1: Figure 5-1 depicts this traditional way of handling containers. The disadvantage of this techniques is a rather low level of isolation as a security issue in the container runtime allows to directly attack the host kernel.

An alternative way of isolating software is virtualization, where software is run on a virtual CPU, provided by a hypervisor. These hypervisors usually rely on specific hardware features to provide comparable performance but still enforce the virtualization boundary. Being a slimmer and more low-level interface, it can be argued that Hypervisor isolation is more secure than traditional container isolation. The right side of Figure 5-1 depicts this setup.

Some platforms and container engines provide additional options to harden the containerized environments. A powerful example is the combination of both presented technologies, where hypervisors are used to enhance the container isolation which is used by some security focused container engines<sup>11</sup>. This is shown in the right side of Figure 5-1.

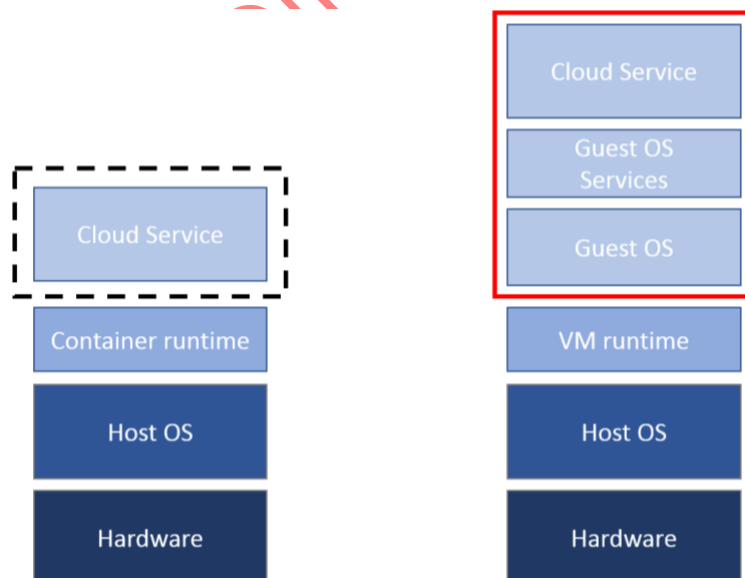


Figure 5-1: Classical micro-service stacks. Containerization on the left side, virtual machines on the right side.

<sup>11</sup> <https://nabla-containers.github.io/>

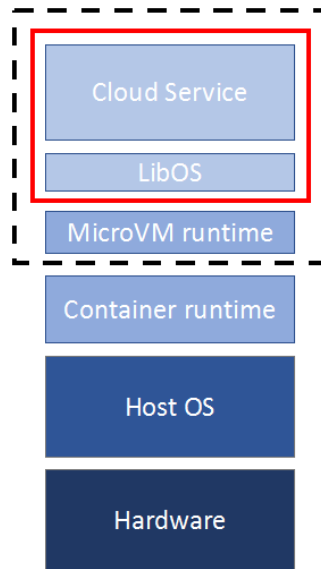


Figure 5-2: Hardened container setup using containers, microVMs and library operating systems.

Despite the acceleration by hardware features, virtual machines increase the overhead. This is mostly due to the fact, that they aim to emulate a real machine as close as possible. In cloud and IoT environments, this is usually not necessary, as an application would only need a small subset of the functionality an operating system and the peripherals can provide and most of them are virtual anyway. Thus, one way of improving the performance is the usage of microVMs, which drop compatibility to classical operating systems, so that the virtual machines can be optimized and shrunk down for increased efficiency. IoT-NGIN focuses on micro-services. Consequently, each container deploys usually one application, which handles e.g., https-requests. In this case, a multi-processor, multi-tasking, and multi-user operating system like Linux as guest operating system in a virtual machine is too generalized. Library Operating Systems (also known as unikernels) are an attractive solution to decrease the overhead. In this case, the kernel is linked as library to the application and realized as bootable application. Utilizing compile-time optimizations, such an image is optimized for a certain type of applications and has a largely reduced attack surface and thus an increased security. In addition, the complete software stack from the kernel through the IP stack to the application itself can be analysed with established compiler techniques. Unneeded code can be removed which reduces the complexity of the system. In IoT-NGIN, RustyHermit is used to show the applicability and robustness of unikernels.

The IoT world isn't only based on Rust. Especially machine learning models, whose support is one of the key objectives of IoT-NGIN are often based on other programming languages. C/C++ is still an important programming language and must be supported by unikernels. RustyHermit can support C/C++ and Fortran by providing a cross-compiler to build unikernels on top of a Linux system. Figure 5-3 shows the runtime system of RustyHermit. The kernel itself is completely written in Rust and named in that figure with *libhermit-rs*. The C library *newlib* is used for other programming languages which build upon the POSIX system interface like C, C++ and others.

The right side of Figure 5-3 shows the concepts of the available execution environments which are the general purpose QEMU and the tailored microVMs Firecracker and Uhyve

mentioned before. Uhyve is developed by the project partners from the RWTH Aachen University and could be improved in this project.

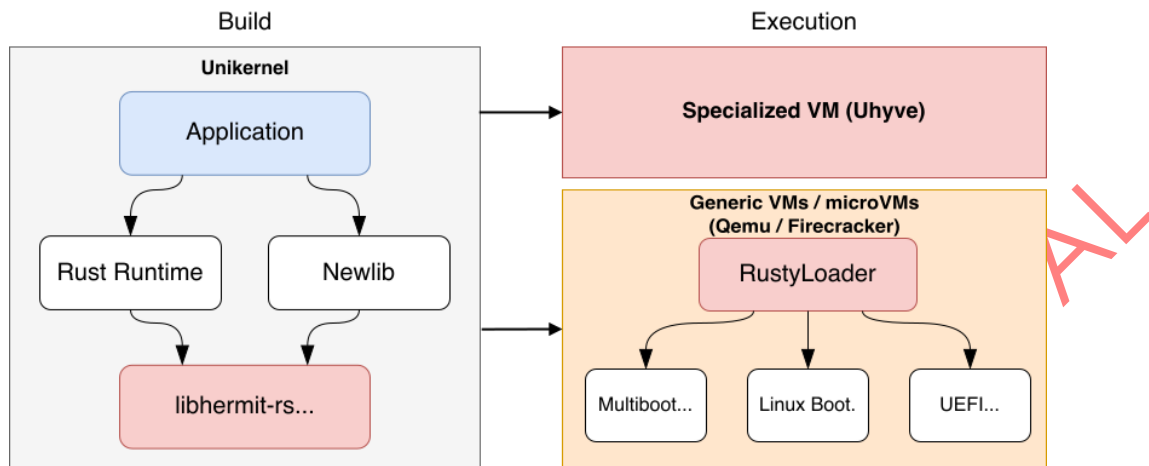


Figure 5-3: Rusty-Hermit conceptual overview

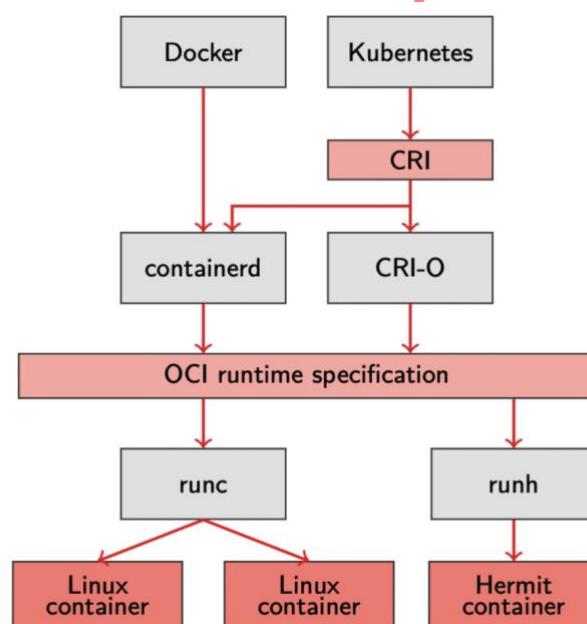


Figure 5-4: The integration of runh in the Docker and Kubernetes software stack

Deliverable 2.2 already explained that the creation of containers standardized, so that other tools like Kubernetes can build upon different varieties of container implementations. The Open Container Initiative (OCI) defines a runtime interface to a tool, which at the end spawns a container. Typically, runc is used by Kubernetes and Docker to spawn containers based on OS-level virtualization. In deliverable 2.2, we defined our own container spawner runh that can spawn common containers but also containers based on a microVM and the unikernel RustyHermit. Figure 5-4 shows how runh integrates into a Kubernetes or Docker setup.

## 5.2 The IoT-NGIN Secure Edge Cloud Framework

The resulting framework developed in this task consists of three major building blocks:

- The improved unikernel RustyHermit
- The container and unikernel runtime runh
- Multiple solutions to execute Machine learning models in the RustyHermit Unikernel

Since the start of the IoT-NGIN Project, 1655 commits were added to the unikernel library libhermit-rs, changing 183 files resulting in 25016 added lines of code and 60875 deleted ones. The rusty-hermit repository had 770 commits in that time, changing 66 files with 3109 added and 2226 removed lines. The microVM Uhyve had 74 files changes with 5585 added and 7505 removed lines of code. Notable changes include the rework of the network stack and debugging support.

New projects in the area of hypervisors like Firecracker use Rust to improve the security behaviour. Firecracker is an alternative to QEMU and is designed to run serverless functions and containers safely. The minimalistic design of Firecracker offers only 5 devices. RustyHermit is now able to run on top of Firecracker. Consequently, the complete software stack is hardened by the usage of Rust on all levels of the software stack. The second component, runh<sup>12</sup>, is a runtime for Kubernetes, which can both run docker containers and unikernels. The ability to choose the enhanced isolation of the unikernels in combination with the compatibility of the vast docker ecosystem is an enabler for unikernels in the micro-service domain.

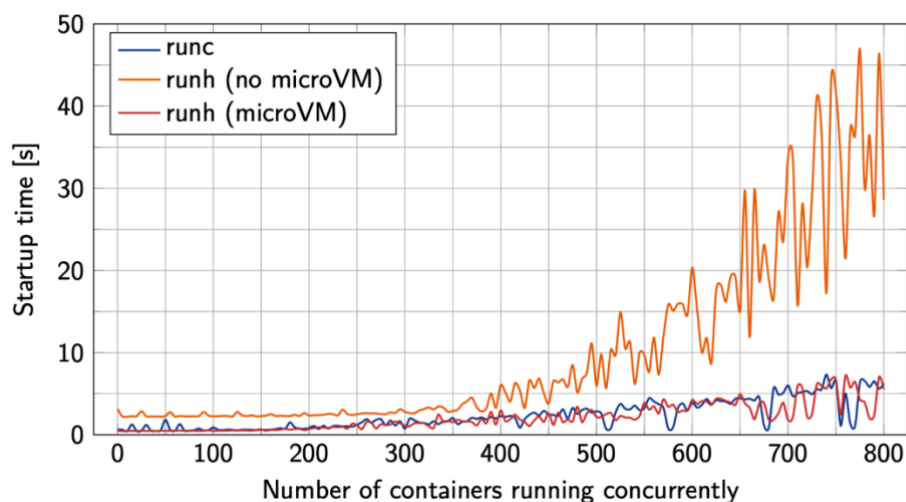


Figure 5-5: Performance comparison of runh and runc using QEMU and microVMs

runh uses microVMs, which promise a lower overhead in comparison to common VMs. The first release uses QEMU as hypervisor to run RustyHermit. QEMU is an established hypervisor especially in Linux environments and offers also a machine model as microVM. By using QEMU, we are able to compare the microVM with a common VM as QEMU is able to support both VM types. The performance gain is depicted in Figure 5-5, which shows the boot time of concurrent containers. The blue line shows the excellent performance of common containers. In comparison to the orange lines shows the performance of common QEMU

<sup>12</sup> <https://github.com/hermitcore/runh>

VMs, the red line shows that the usage of QEMU-based microVMs<sup>13</sup> provides nearly the same performance as common containers, whilst providing a stronger isolation at the same time.

The third component - the machine learning support - is explained in more detail in the following section.

## 5.3 Application to ML optimization in IoT-NGIN

There is an increasing usage of intelligence in IoT based applications, that use the application context, captured by IoT sensors or devices, to make smart decisions. This intelligence is generated by the inference capabilities of ML models that have been trained with that context data. However, the inference is required, in many situations, to make near-real-time decisions, leveraging the high performance of those ML models. Although ML models can be trained offline in Cloud infrastructures, their inference process is mostly executed in Edge clusters, near the context data used as input, in order to reduce the data access latency, to speed up the overall inference process. In general, different strategies, including software and algorithm optimizations, hardware acceleration (GPUs, TPUs, FPGAs, etc), have been proposed for ML inference optimization, also for the Edge [Ref14].

In the context of unikernels for secure execution, concrete choices of hardware (e.g., CPU) and software (e.g., C/C++/Rust) may constrain the execution. However, some performance gain is expected when inference is executed in unikernels, with binaries compiled with those compatible languages (i.e., C/C++), when compared with the execution with other popular languages among data scientist, such as Python, which offer significant lower performance on CPUs.

This results in the following requirements such a solution must have for the application in the Edge-Cloud Framework:

- It must produce binary, self-consistent and not dynamically linked, CPU-compatible programs, so that they can be executed in the unikernel
- They must be compatible with the ML frameworks used in IoT-NGIN, such as PyTorch and TensorFlow.

The first requirement enforces the usage of ML frameworks that offers either C/C++ bindings or libraries, as those programming environments may produce self-consistent binaries that do not require of a specific programming execution environment (as Python does). However, for most of today's frameworks, various solutions to achieve this goal exist. In combination with the second requirement the following approaches IoT-NGIN ML inference execution within unikernels are:

- Apache TVM compilation<sup>15</sup>
- Torch C++<sup>3</sup>
- TensorFlow C++<sup>4</sup>
- TensorFlow C API<sup>5</sup> with CppFlow<sup>6</sup>
- ONNX Runtime<sup>7</sup>

<sup>13</sup> <https://qemu.readthedocs.io/en/latest/system/i386/microvm.html>

<sup>14</sup> Shuvo, M. M. H., Islam, S. K., Cheng, J., & Morshed, B. I. (2022). Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review. *Proceedings of the IEEE*.

<sup>15</sup> <https://tvm.apache.org/>

We'll briefly describe these different approaches:

Apache TVM (for Tensor Virtual Machine) describes itself as an "open-source machine learning compiler framework for CPUs, GPUs, and machine learning accelerators". Depending on the definition, it could also be called a transpiler, as it can transform ML models for various execution backends like GPUs or DSPs, but also into executable C-code, which makes it interesting for Unikernels. TVM can transform multiple input formats, including TensorFlow and PyTorch.

Torch C++ API is easy to install from its portal.<sup>8</sup> It offers binary downloads for installation, so its compilation from sources is not required. Its documentation is quite good and it is well supported by the community forums.

TensorFlow C++ is not available in binary format for direct download and installation, so it must be compiled from sources before being installed. However, the compilation process is hard, although well documented, imposing constraints on the compatible versions of the gcc compiler, the Python development environment and the Bazel building tool. As the compatible versions of the gcc compiler are not among the newest ones present in most recent versions of Linux OS, its compilation could be challenging on most machines. In the community, few users have faced successfully these compilation challenges and offered some Docker containers to build TensorFlow from sources<sup>9</sup>. TensorFlow C++ is not well documented, and scarce support is found in Internet communities.

TensorFlow C API is a C binding for TensorFlow. There are binary downloads for popular OS that are easy to install. However, it is not well documented and its usage in C++ programs could be challenging.

To overcome these limitations, CppFlow offers C++ headers to bind the TensorFlow C API and produce C++ binaries for TensorFlow inference, without requiring to compile TensorFlow C++ from sources with Bazel.

ONNX Runtime is an optimized ML inference (and training) execution platform, that offers API binding for several programming languages (e.g., C/C++, Python, Java, etc), on multiple hardware (X64/X86, ARM, etc) and acceleration (CUDA, TensorRT, etc). ONNX Runtime executes ML inference on ONNX models, which are intermediate model created from models build with other ML backends (e.g., PyTorch, TensorFlow, etc).

We have been evaluating the mentioned technologies<sup>16</sup>. Initially, we experimented with some concrete examples we found in the communities, then we applied those experiences to some ML inference examples for image classification, based on Alexnet and Resnet18. Next, we applied them to some of the IoT-NGIN Living Labs use cases, namely: the power generation forecasting on the Smart Energy LL, and the sensor detection on the Smart Agriculture LL (see D6.2 for further details on these use cases).

In the following, we present the TVM approach, as well as the Torch C++ approach, as these illustrate two rather different approaches to the problem, the other approaches follow the same principles.

---

<sup>16</sup> [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/edge-cloud-framework/ml-inference-examples-for-unikernels/-/tree/experiments](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/edge-cloud-framework/ml-inference-examples-for-unikernels/-/tree/experiments)



### 5.3.1 Machine learning support using TVM cross compilation

Apache TVM is a framework to optimize and execute different formats of machine learning models on various hardware architectures and accelerators. It works as a compiler that can read multiple model formats, interprets, and processes them as intermediate representation and generates code for the selected backend. The execution of TVM processed models in RustyHermit is one of the components of the developed framework.

The source code is published at the project's Gitlab Repository<sup>17</sup> illustrates the usage and acts as a starting point for other projects. The most relevant part of the project is the CMakeLists.txt file, which invokes TVM and let it generate the C code for the unikernel. The invocation of TVM looks as follows:

```
python -m tvm.driver.tvmc compile --target=c --runtime=crt --executor=aot
--executor-aot-interface-api=c --executor-aot-unpacked-api=1
--pass-config=tir.disable_vectorize=1 --output-format=mlf
--output=<BINARY_DIR>/module.tar <DOWNLOADED_FILE>
```

This generates the source files `_codegen/host/src/default_lib0.c`, `codegen/host/src/default_lib1.c` and `runtime/src/runtime/crt/common/crt_backend_api.c` containing the executable model. In this example, the *MobileNet v2 1.0 224 INT8* image classification model is automatically downloaded and used. Additionally, the python script `convert_labels.py` generates the class labels for the output vector and the `convert_images.py` script converts the input image into a raw c-code file. These steps are necessary in this minimal example, a real -world application would adapt this depending on their needs.

TVM is a great way of running static ML models in the edge-cloud framework. However, for adaptive or more advanced models, Torch C++ is better suited.

### 5.3.2 Torch C++ ML Models in Unikernels

In the following section, we describe the application of the Torch C++ approach for the execution of ML models withing the Framework. First, we briefly explain the models, the application in the Unikernel, two experiments have been conducted with Torch C++:

- An Alexnet image classification,
- A Smart Energy LL power generation forecasting

#### 5.3.2.1 Alexnet classification

The source code for this experiment is available at the project's Gitlab Repository<sup>18</sup>. This is an image classification inference problem that uses a pre-trained Alexnet model (obtained from TorchVision) to classify an input image (a Labrador dog) among 1000 classes.

---

<sup>17</sup> [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/edge-cloud-framework/hermit-tvm](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/edge-cloud-framework/hermit-tvm)

<sup>18</sup> [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/edge-cloud-framework/ml-inference-examples-for-unikernels/-/tree/experiments/torch/alexnet](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/edge-cloud-framework/ml-inference-examples-for-unikernels/-/tree/experiments/torch/alexnet)

The C++ code (`./src/alexnet-app.cpp`) uses cmake for building and the libraries Torchlib C++ and OpenCV for ML inference and image pre-processing, respectively. The source code includes a `readme.md` file that describes the procedure to install the dependencies, build and execute the experiment. The inference results on a list of best predictions with scores:

*Prediction: whippet with probability: 50.417*

*Prediction: Saluki, gazelle hound with probability: 30.0007*

*Prediction: Ibiza hound, Ibiza Podenco with probability: 5.98667*

*Prediction: Labrador retriever with probability: 5.31794*

...

Performance gain w.r.t. the original Python execution is about 54% in our experiments with CPU.

### 5.3.2.2 Smart Energy LL power generation forecasting

The source code of this experiment is available in the same repository<sup>19</sup>. This is a time series forecasting problem that uses a GRU-base regressor (a variant of LSTM) to forecast the future power generation of an electric grid (EG). This model is online trained with MLaaS Online Learning service (see D3.3 for further details). In this experiment we use a snapshot of the trained model, retrieved from the MLaaS Model Storage, where it is stored upon gains on performance.

This model takes, as input, a tensor with the last 36 hourly measurements of the EG power generation, and provides, as output, a prediction for the next hour power generation.

The C++ code (`./src/power_generation_forecast.cpp`) uses cmake for building and the Torchlib C++ library for the ML inference. The source code includes a `readme.md` file that describes the procedure to install the dependencies, build and execute the experiment.

The inference outputs the prediction of the next hour power generation:

*Power generation prediction: 0.890847*

Performance gain w.r.t. the original Python execution is about 65% in our experiments with CPU.

Both experiments showcase the performance gains and the ability to produce unikernels-compatible binaries for Torch based inference in a range of different applications (classification, regression).

### 5.3.2.3 Unikernel execution

The RustyHermit cross-compiler<sup>20</sup> adheres typical standards to build larger C++ applications. It is based on gcc 7.5.0, offers an ANSI C library, the POSIX interface for thread handling, and adheres the C++14 standard. However, PyTorch must be modified to support RustyHermit. RustyHermit's C library was originally designed for embedded systems. To reduce the

---

<sup>19</sup> [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/edge-cloud-framework/ml-inference-examples-for-unikernels/-/tree/experiments/torch/power\\_generation\\_forecast](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/edge-cloud-framework/ml-inference-examples-for-unikernels/-/tree/experiments/torch/power_generation_forecast)

<sup>20</sup> <https://github.com/hermitcore/hermit-toolchain>



overhead, the C library doesn't offer all features of the GNU C library. Therefore, we must configure the PyTorch C++ runtime in the same manner as it is done for mobile platforms as e.g., iOS or Android, as these operating systems provides less features and thus don't require features only present in the GNU C library.

The second challenge is the adaption of all components, which handle threads. RustyHermit itself supports POSIX threads, but PyTorch's source code and its helper library Kineto is only designed to run directly on top of Linux, Windows, macOS and their branches (e.g, Android) and uses processes instead of threads. Consequently, the source code must be modified to use the Pthread interface.

Overall, the build system and C++ source code of PyTorch and Kineto are modified to use already existing code, which was originally designed for Android or Pthread-based operating systems. In total 31 files are modified with 305 insertions and 11 deletions. In comparison to the complexity of the whole PyTorch project with all helper library (10 million lines of code within 39737 files) are these changes minimal.

Only the helper library `cpuinfo` provides new code for RustyHermit. `cpuinfo` is used to determine the number of processors and their cache sizes. The library depends on operating system interfaces, which must be adapted to support RustyHermit. However, the complexity of the code is small (4 files, 275 insertions). We will submit all changes to the open-source projects so that future projects can also benefit from them.

The source code of the modified libraries is available at:

[https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/edge-cloud-framework/pytorch](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/edge-cloud-framework/pytorch)

[https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/edge-cloud-framework/kineto/-/tree/hermit](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/edge-cloud-framework/kineto/-/tree/hermit)

[https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_underlying\\_technology/edge-cloud-framework/cpuinfo/-/tree/hermit](https://gitlab.com/h2020-iot-ngin/enhancing_iot_underlying_technology/edge-cloud-framework/cpuinfo/-/tree/hermit)

The power generation forecast experiment (see section 5.3.2.2) is used to compare a unikernel with a common Linux application. To run the unikernel within a virtual machine, the microVM Uhyve is used. The RustyHermit foolchain builds a single image, which is bootable within any common VM. The image has a size of 89 (62 stripped) MByte. The Linux application has only a size of 153 KByte. But in comparison to the Unikernel, the Linux application has to load at runtime shared libraries with a total size of 480 (405 stripped) MByte which is more than five (six for stripped binaries) times higher. This is, because a dynamic linker can't remove unneeded code, as also other applications could use the same shared library. This shows the benefits of a static code analysis, where the linker is able to optimize the applications and to remove unneeded code. In cloud environments, where often only one application (micro service) is running in a container, the advantages of code sharing do not come into play. Common containers share the host kernel, but every container provides their own set of shared libraries.

In addition, the unikernel is 25% faster in comparison to the Linux applications. because the overhead of loading the shared libraries at runtime does the Linux application never compensate. This huge speed up is of course not representative for all applications and further analysis is needed to better understand runtime behaviour.

The Alexnet classification (see section 5.3.2.1) is based on Pytorch and also OpenCV, which is used for ML inference and image pre-processing. To use OpenCV for Alexnet, OpenCV

must be cross compiled for RustHermit. The number of changes is clearly smaller in comparison PyTorch, but the kind of changes are similar. Mainly the configuration files need to be modified. In total, 11 files are changed with 11 insertions and 9 deletions. In comparison to OpenCV's 2 million lines of code, these changes are also rather insignificant.

The size of the unikernel increase in comparison to the power generation forecast to 104 MByte (75 MByte stripped). OpenCV seems to be highly optimized for Linux. A deeper analysis is required to understand the behaviour completely. But the memory consumption is clearly high in comparison to RustyHermit, as at least PyTorch (>400 MByte) and a few components of OpenCV have to be loaded into memory.

These first results show that the microVM Uhyve has an excellent start-up time and the unikernel toolchain doesn't introduce new additional overhead. With the benefit of a low memory footprint and a stronger isolation from the host system, unikernels could be a scalable and robust architecture for cloud environments.

## 5.4 Conclusions

In this chapter we explained the different aspects of the secure execution framework that uses unikernels for micro-services. This framework builds upon Kubernetes, which allows the application in cloud and edge-cloud environments. The use of unikernels and micro-VMs provides a high isolation whilst maintaining high performance and thus scalability. The application of this framework was shown by running different approaches of ML inference using this technology. The various experiments have been described and the source is available on the project's repository. The results show that ML models have a much smaller footprint and can execute even faster on the unikernel, which in combination with the higher isolation makes this technology an interesting choice for cloud/edge-cloud environments. Future work on this topic is the adoption of all the approaches for ML-support in the framework as well as running more models on the framework to explore all possible edge-cases.

## 6 How the technology is transferred and demonstrated in use cases and living labs

The deployment of CMC 5G Core including the TSN and 5GLAN functionality has been installed and tested in ABB living lab. In the testing CMC acquired latest 5G device that includes support for Ethernet PDU. The installation includes the latest release of CMC 5GC with support for Ethernet PDU. However, the ABB living labs include a different 5G base station compared to the one used in CMC laboratory. Thus, the base station in ABB living labs did not recognize the Ethernet PDU session from the 5G device and the connection was terminated. This led to failure of time synchronization process so the protocol taking care of time synchronization that works over Ethernet was not completed successfully. However, the same installation in CMC lab where there was the possibility of accessing the settings of the base station did allow to have an Ethernet PDU session which resulted in successful synchronization.

DRAFT - PENDING EC APPROVAL

## 7 Conclusions

This chapter will provide the conclusions on the four different enhancements for IoT underlying technology discussed in this deliverable.

Starting with the enhancement of 5G coverage. The deliverable discussed the different possibilities for extending the device coverage. The focus was put on the utilization of UEs for relaying the communication and thus enabling devices that otherwise would not have cell coverage to still connect to cell services. Two types of communication were evaluated. The first examined the possibility to extend the network based on 5G relaying and the second WIFI direct communication. Both are compared based on throughput and RTT measurements. The 5G approach has an average RTT of about 23 ms and an average throughput of up to 115 Mbps. Compared to this the Wifi Direct connection achieves a average RTT of also about 23 ms and an average throughput of up to 140 Mbps. Depending on the available UE and hardware configuration this shows a flexible solution for the coverage extension.

The second major topic of this deliverable evaluates the possibility of deterministic communication. This is achieved by utilizing a custom 5G core. The measurements provided in that chapter show the time synchronization of the UE compared to a network without time sensitive networking feature.

In the third chapter the developments concerning the 5G Resource Management API are described. The final version of the proposed simplified API is shown and explained. Furthermore the adapters for container control with Kubernetes, runh and for the slice creation and service control are described. The implementation is reduced to minimal examples as this is meant to be a proof of concept. The simplification can be seen with the calls needed for the slice manager. Here on call for starting a service results in four different calls to the slice manage backend. Since it was not possible to test the behaviour against the real hardware a additional simulator was implemented this software behaves like the real API and is based on the official open API specification for the slice manager. This shows the possibility for rapid prototyping without the final software stack up and running. For the unikernel and Kubernetes adapter it is shown that a single command for starting a webserver can be deployed on the specific infrastructure depending on the used software stack.

Lastly, the Secure Edge Cloud Framework was presented, which introduces the novel unikernel technology to the cloud and edge-cloud domain. This approach provides a higher isolation for microservices through the use of virtual machines. By using microVMs, the overhead is kept low and is comparable to the widely used container technology. One of the primary targets for microservices in IoT-NGIN are machine learning applications, thus the framework includes several methodologies to execute inference on unikernels in cloud applications. The chapter demonstrates some of the most relevant ones.

The demonstration of TSN and 5G-LAN in the ABB living lab could not be completed so far, but the process is still ongoing and successful results are expected during the remainder of the project.

The presented technologies all have their own scope, but are all expected to have an impact in their own domain and do their part in shaping the future of the IoT landscape.