Department of Information Science and Technology

A Software-Defined Network Solution for Managing Fog Computing Resources in Sensor Networks

A Dissertation presented in partial fulfillment of the Requirements for the Degree of Master in Telecommunications and Computer Engineering

by

Patrícia Galego Cardoso

Supervisors:

Prof. Dr. Rui Miguel Neto Marinheiro, ISCTE-IUL

Prof. Dr. José André Rocha Sá Moura, ISCTE-IUL

LISBON, OCTOBER 2019

Department of Information Science and Technology

A Software-Defined Network Solution for Managing Fog Computing Resources in Sensor Networks

A Dissertation presented in partial fulfillment of the Requirements for the Degree of Master in Telecommunications and Computer Engineering

by

Patrícia Galego Cardoso

Supervisors:

Prof. Dr. Rui Miguel Neto Marinheiro, ISCTE-IUL

Prof. Dr. José André Rocha Sá Moura, ISCTE-IUL

LISBON, OCTOBER 2019

This page was intentionally left blank

No matter how many mistakes you make
or how slow you progress, you are still way
ahead of everyone who isn't trying
Anthony Robbins

Acknowledgments

Completing my master's degree was an important achievement for me and I would like to express my never-ending gratitude to all who made it possible.

First and foremost, I offer my sincere gratitude to my supervisors, Professor José Moura and Professor Rui Marinheiro, for all the support outlining and executing the work. Their knowledge, availability and patience were crucial to the realization of this thesis.

A special thank you to Instituto de Telecomunicações (IT) for having me thorough the execution of this thesis.

I take this opportunity to thank all my colleagues from CGI, for providing me with such a friendly and supportive environment. I am thankful for everything I've learned from them and for all the encouragement whilst finishing my degree.

To my friend Rita, since I am one of the luckiest people in the world to have you as a friend. You have looked after me, supported me and I thank you so much.

Above all, I would like to thank my parents for always encouraging me in every single decision, this one included. To my sisters, Márcia and Joana, for being my shelter and support. Thank you!

This page was intentionally left blank

Abstract

The fast growth of Internet-connected embedded devices raises new challenges for the traditional network design, such as scalability, diversity, and complexity. To endorse these challenges, this thesis suggests the aggregation of several emerging technologies: software-defined networking (SDN), fog computing, containerization and sensor virtualization.

This thesis proposes, designs, implements and evaluates a new solution based on the emergent paradigm of SDN to efficiently manage virtualized resources located at the network edge in scenarios involving embedded sensor devices. The sensor virtualization through the containers provides agility, flexibility and abstraction for the data processing, being possible to summarize the huge amount of data produced by sensor devices. The proposed architecture uses a software-defined system, managed by a Ryu SDN controller, and a websocket broker written from scratch that analyses the messages sent to the controller and activates containers when required.

Performance and functional tests were performed to assess the time required from activating the sensor containers to being able to communicate with them. The results were obtained by sending four ICMP packets. The best time response results were obtained by the proactive controller behavior mode, when compared to the hybrid and reactive modes.

This thesis contributed to fill the gaps in the area of IoT or sensor networks, concerning the design and implementation of an architecture that performed on-demand activation of offline IoT fog computing resources by using an SDN controller and sensor virtualization through containers.

***Keywords*:** Software-defined Networking, Internet of Things, Fog Computing, Linux Container

This page was intentionally left blank

Resumo

O rápido crescimento de dispositivos embebidos conectados à Internet gera novos desafios para a arquitetura de rede tradicional, tais como escalabilidade, diversidade e complexidade. Para resolver estes desafios, esta tese sugere a agregação de diversas tecnologias emergentes: rede definida por software (SDN), contentores, computação na periferia e virtualização de sensores.

Esta tese propõe, projeta, implementa e avalia uma nova solução baseada no paradigma emergente do SDN para gerir, de forma eficiente, recursos virtualizados que se localizam na periferia da rede, em cenários com sensores embebidos. A virtualização de sensores, através do uso de contentores, fornece agilidade, flexibilidade e abstração para processamento de dados, sendo possível a sumarização do grande volume de dados produzido pelos sensores. A arquitetura proposta usa um sistema definido por *software*, gerido por um controlador SDN Ryu, e um *websocket* broker escrito desde o zero, que analisa as mensagens enviadas ao controlador e ativa contentores quando necessário.

Foram realizados testes funcionais e de desempenho de forma a ser possível avaliar o tempo necessário desde a ativação de um contentor de sensores até ser possível a comunicação com este. Os resultados foram obtidos através do envio de quatro pacotes ICMP. O melhor resultado foi obtido pelo modo de comportamento proativo do controlador, quando comparado aos modos híbrido e reativo.

Esta tese contribuiu para preencher as lacunas na área de IoT ou redes de sensores, no que diz respeito ao desenho e implementação de uma arquitetura que executa a ativação sob pedido de recursos computacionais e periféricos de IoT quando estes se encontram desligados, através do uso de um controlador SDN e virtualização de sensores através de contentores.

***Palavras-chave*:** Redes definidas por software, Internet das coisas, Computação na periferia, Contentores Linux

This page was intentionally left blank

Nomenclature and abbreviations

API	Application Programming Interface
AHP	Analytical Hierarchy Process
ARP	Address Resolution Protocol
BGP-LS	Border Gateway Protocol Link-State
CoAP	Constrained Application Protocol
CPU	Central Process Unit
DTLS	Datagram Transport Layer Security
EPL	Eclipse Public License
ETSI	European Telecommunications Standards Institute
GUI	Graphical User Interface
GW	Gateway
HP	Hewlett-Packard
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IBM	International Business Machines
ICMP	Internet Control Message Protocol
IDC	International Data Corporate
IETF	Internet Engineering Task Force
IoT	Internet of Things
IoTDM	IoT Data Management
IP	Internet Protocol
JSON	JavaScript Object Notation
LACP	Link Aggregation Control Protocol
LISP	Locator/ID Separation Protocol
LXC	Linux Container
M2M	Machine-to-Machine
MAC	Media Access Control
MB	Megabyte
MCDM	Multi-Criteria Decision Making
MQTT	Message Queue Telemetry Transport
MQTT-SN	Message Queue Telemetry Transport for Sensor Networks

ms	Millisecond
NBI	Northbound Interface
NETCONF	Network Configuration Protocol
NFV	Network Function Virtualization
ODL	OpenDaylight
OF-CONFIG	OpenFlow Management and Configuration Protocol
ONF	Open Networking Foundation
OS	Operating System
OSGI	Open Services Gateway Initiative
OTP	Open Telecom Platform
OVS	Open vSwitch
QoS	Quality of Service
REST	Representational State Transfer
SB	Southbound API
SCTP	Stream Control Transmission Protocol
SDK	Software Development Kits
SDN	Software-defined Networking
SNMP	Simple Network Management Protocol
SSL	Secure Socket Layer
SVM	Sensor Virtualization Module
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
URI	Universal Resource Identifier
VM	Virtual Machine
WAN	Wide Area Netwoks
XML	Extensible Markup Language

This page was intentionally left blank

Table of Contents

Acknowledgments VII

Abstract IX

Resumo XI

Nomenclature and abbreviations XIII

Table of Contents XI

List of Figures XI

List of Tables XII

Chapter 1 – Introduction 1

Chapter 2 – Literature Review 4

2.1. Internet of Things 4

2.1.1 IoT Platforms 5

2.1.2. IoT Communication Protocols 6

2.1.2.1. MQTT and MQTT-SN Protocols 6

2.1.2.2. CoAP 8

2.1.2.3. Comparison MQTT, MQTT-SN and CoAP 9

2.2. Virtualization 10

2.2.1. Hardware Virtualization 10

2.2.2. Operating System – Level Virtualization 11

2.2.3. Network Sensor Function Virtualization 15

2.3. Fog and Edge Computing 17

2.4. Software-Defined Networking 20

2.4.1. SDN Architecture 20

2.4.1.1. Network Elements 21

2.4.1.1.1. Controlling Example 24

2.4.1.2. SDN Controllers 24

2.4.1.2.1. OpendayLight 24

2.4.1.2.2. RYU 25

2.4.1.2.3. Comparison Amongst Several SDN Controllers 26
2.4.1.3. Northbound Management 27
2.4.2. Software-Defined Networking Virtualization 28
2.5. Related Work 29
Chapter 3 – Proposed Solution 32
3.1. Architecture 32
3.2. Deployment 33
Chapter 4 – Results and Discussion 39
4.1. Performance Tests 40
4.2. Functional Tests 47
Chapter 5 - Conclusions and Future Work 49
5.1. Conclusions 49
5.2. Future Work 50
Chapter 6 – References 51

List of Figures

Figure 1 - Traditional networking vs SDN [7] 1
Figure 2 - IoT Architecture [14] 4
Figure 3 - MQTT-SN Architecture [20] 7
Figure 4 - Transparent and Aggregating Gateways [20] 8
Figure 5 - CoAP 8
Figure 6 - Containers vs Virtual Machines [27] 11
Figure 7 - LXC Architecture 12
Figure 8 - Docker Architecture [31] 13
Figure 9 – ETSI Network Function Virtualization Architectural Framework [33] 15
Figure 10 - Sensor virtualization module architecture and usage scenario [36] 16
Figure 11 - Sensor data mash-up [36] 16
Figure 12 – Hierarchical architecture of fog computing [3] 17
Figure 13 – The architecture of edge computing [3] 18
Figure 14 – Comparison of fog computing and edge computing [39] 19
Figure 15 - General architectural framework for SDN [23] 21
Figure 16 - Main components of an OpenFlow switch [46] 22
Figure 17 - The operation of SDN (controller - switch) [5] 24
Figure 18 - Ryu Framework [55] 26
Figure 19 - IoT SDN/NFV Architecture [66] 31
Figure 20 - SDN architecture for IoT based on Fog computing [70] 31
Figure 21 - Proposed Architecture 32
Figure 22 – Detailed proposed architecture 34
Figure 23 - Network Topology Configuration 34
Figure 24 - Network Topology 35
Figure 25 - Messages exchanged between Open vSwitch and Ryu Controller in the reactive mode 35
Figure 26 – Sequence diagram for hybrid mode 36
Figure 27 – Hexadecimal message and analysis 37

Figure 28 - MQTT-SN Scenario 38
Figure 29 - Architecture topology 39
Figure 30 - Reactive Mode - Scenario 1 and 2..... 41
Figure 31 - Packet-In and Packet Out ARP Messages (Scenario 2) 41
Figure 32 - ICMP traffic in the reactive mode (Scenario 2) 42
Figure 33 - Packet-In and Packet Out ICMP Messages (Scenario 2)..... 42
Figure 34 – Reactive Mode - Ping from LXC3 (control) to LXC1 (sensorTemp) and LXC2 (sensorHum) 42
Figure 35 – Hybrid Mode - Scenario 1 and 2, 1st packet..... 43
Figure 36 - Hybrid Mode - Scenario 1 and 2, 2nd to 4th packet..... 44
Figure 37 - Hybrid Mode - Ping from LXC3 to LXC2 and switch flow table rules..... 44
Figure 38 - Proactive Mode - Scenario 1 and 2, 1st packet..... 45
Figure 39 - Proactive Mode - Scenario 1 and 2, 2nd to 4th packet..... 46
Figure 40 - Message exchanged through MQTT-SN (LXC3 first publishes message)..... 47
Figure 41 - Message exchanged through MQTT-SN (LXC1 first subscribes topic)..... 48

List of Tables

Table 1 – Comparison between MQTT, MQTT-SN and CoAP [21] 9
Table 2 – Containers and Virtual Machines features comparison [25][26] 11
Table 3 – Container Implementation Comparison [25]..... 14
Table 4 – Similarities and differences between fog and edge computing [3]..... 20
Table 5 - Feature-based comparison of SDN controllers [42]..... 27
Table 6 - List of Related Research..... 29
Table 7 - Performance Tests Comparison 46

Chapter 1 – Introduction

1.1. Motivation and Background

The concept of Internet of Things (IoT) was used for the first time in 1999 by Kevin Ashton, merging the human's communication network and the real world of things, such as devices, sensors and actuators, that are linked together and connected to the Internet, having the ability to collect and transmit environment information [1]. According to IDC (International Data Corporate), it is estimated that by the end of 2020 there will be 212 billion of things connected to the Internet [2], thanks to the improvement of the telecom sector, the cheaper Internet and the facility to produce smaller but more powerful hardware. This exponential increase of IoT devices create challenges in terms of scalability, mobility, heterogeneity and security. The enormous scale of devices connected, the management of the high volume of data produced, the use of different communication protocols and different hardware and software by manufacturers makes the use of traditional network architecture ineffective, being required a more flexible and dynamic network's architecture.

Transferring all the IoT data to the cloud is also becoming a challenge, requiring a high bandwidth and energy consumption. The addition of a fog layer between IoT devices and the cloud would increase the performance, mobility and security, and reduce the data volume exchange and the latency [3]. However the addition of this technology introduces some complexity and mobility issues.

Software-defined networking (SDN) has emerged as a network architecture that allows the management of the complexity of Fog Computing environment and helps solving the IoT heterogeneity problem, enabling the creation of independent features and protocols of manufacturers, overcoming the problems related to the closed hardware and proprietary software [4]. In traditional networks, the control plane and the data plane are located within the network elements, requiring a configuration on each device, using a low-level and often vendor-specific commands. SDN, unlike the traditional network, and, as shown in Figure 1, it separates the control plane from the data plane, having a centralized control that provides an abstract overview of all the network topology [5]. Thus, it is possible to optimize traffic management as well as support service requirement from a centralized user interface (UI), offering greater agility, traffic programmability and the capability to implement network automation [6].

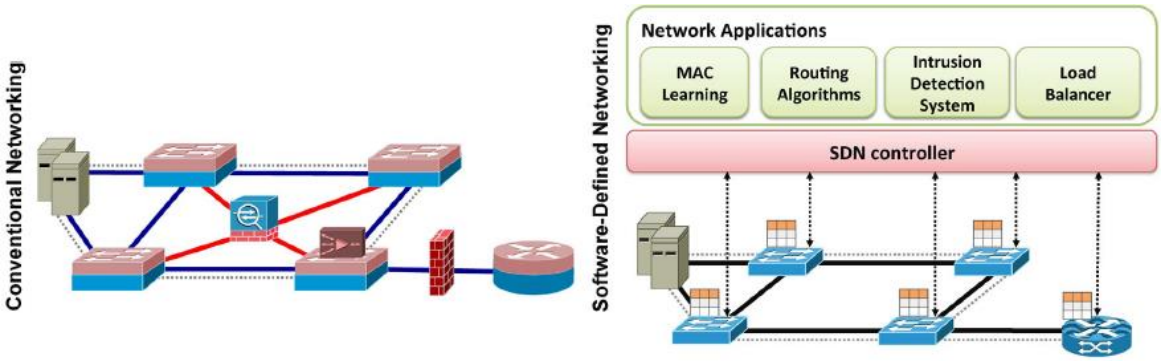

Figure 1 - Traditional networking vs SDN [7]

The use of virtualization techniques applied to SDN and sensors can help facing the scalability and heterogeneity issues. The use of sensors virtualization provides software abstraction, reducing the number of physical devices needed and allowing their resources management through open APIs [8][9].

1.2. Research Questions and Objectives

Considering the above-mentioned gaps in the area of IoT or sensor networks and the opportunities that SDN brings, there is a relevant question:

- Is it possible to efficiently perform on-demand activation of IoT computing resources using a software-defined system?

As such, the main goal of this thesis consists on proposing an architecture that uses a software-defined networking approach to efficiently activate fog computational resources on-demand, as required by IoT or sensor networks applications.

1.3. Investigation Method

After identifying the problem and objectives and to proceed to the investigation proposed on this thesis, an architecture was designed and evaluated, and the obtained results analyzed. This architecture is composed by a virtual switch that communicates to a software-defined networking controller through the southbound API. The controller communicates through the northbound API with an intelligent Broker that was created to analyze messages and to request a container on-demand.

To test the efficiency of the proposed solution, performance and functional tests were performed. The performance test was executed with three different SDN controller code versions to understand the time required from activating the containers to being able to communicate with them. The functional test consisted in the communication between containers through a very popular sensor protocol, i.e. MQTT-SN. The final goal of this thesis in terms of design and implementation was the application of both tests in simultaneous, *i.e.* the activation of containers and the immediate messages exchange through the MQTT-SN protocol.

Overall, this thesis helps facing the IoT and sensor networks issues concerning the design and implementation of an architecture that performed on-demand activation of offline IoT fog computing resources by using an SDN controller, containerization and sensor virtualization.

1.4. Dissertation Outline

The rest of this thesis is organized as follows. Chapter 2 discusses the literature review and describes the different technologies used. Chapter 3 details the proposed solution, where the proposed architecture and the implementation can be found. Chapter 4 discusses the obtained results. Finally, the Chapter 5 presents the general conclusions about the work developed, as well as suggestions of future work.

Chapter 2 – Literature Review

2.1. Internet of Things

The Internet of Things (IoT) is a global network where things, like sensors, devices and actuators, are connected to the Internet, sensing and collecting data, allowing the communication not just between humans and things but also between humans themselves.

The IoT devices possess rigorous requirements in terms of hardware and software, making these devices intelligent and autonomous, capable of performing tasks of detection, monitoring and interaction with the environment. Their capability for collecting data allow these devices to change their state as well as their context dynamically. The connectivity amongst IoT devices is essential as it contributes for the global intelligence of the IoT network. However, this connectivity creates a challenge in terms of the enormous scale of devices connected and the management of the high volume of data produced, being required the use of a network capable of adapting in an efficient and immediate way [10][11]. Alongside with scalability, IoT devices heterogeneity and security issues must also be taken into consideration. IoT devices require security features to ensure, amongst others, authentication, confidentiality, privacy, access control, anonymity and availability. With the intend of IoT to connect devices from different vendors, with different characteristics and complexity, it is necessary the implementation of protocols that support heterogeneous environments as well as security features [12][13].

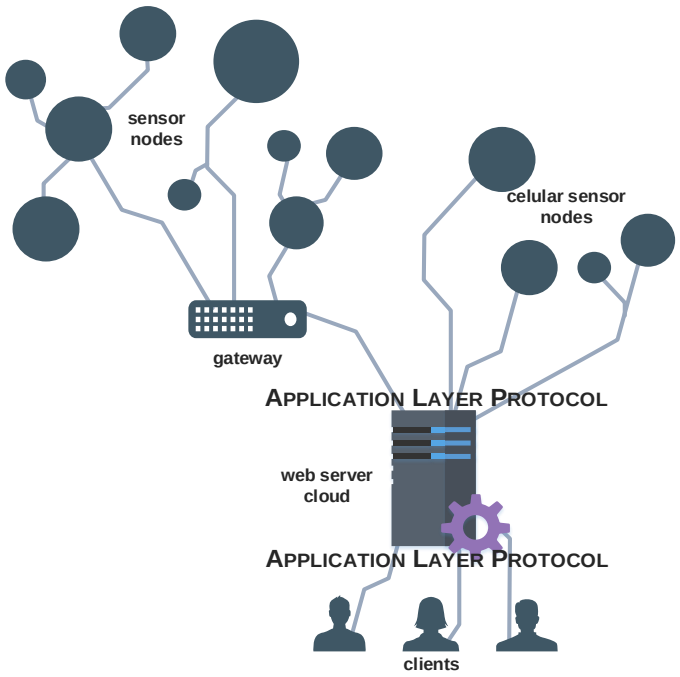

Figure 2 - IoT Architecture [14]

2.1.1 IoT Platforms

Due to the scalability problem created by the Internet of Things, it is necessary the use of platforms that provide communication protocols in order to connect the IoT devices to the network. These platforms enable the connectivity, services and cloud for the devices and can be integrated with smartphones allowing a real-time portable monitoring, optimizing costs and ensuring safety and reliability [15].

According to IoT Analytics [16], there are more than 300 IoT platforms, which can be divided based on its level of technology depth. The first level is composed by the connectivity platform, that simply collects the data from the devices while the second level refers to the action platforms, which manages the connection and provides trigger actions based on specific events. Full-scale platforms are in the third level of technology depth. These are the most advanced platforms that besides allowing connectivity and action, also enable external interfaces such as APIs, SDKs and gateways, support various protocols and standards and provide database solutions. Although this last level is the one with most features, the major percentage of IoT platforms provided by companies are at level 1.

An IoT platform should be open, flexible, providing a perfect integration between heterogeneous devices and using all kind of topologies and protocols. In addition, the IoT platform should be easy to use and ensure software updates, provide storage for a huge data volume and a fast execution of actions based on specific sensor data. In order to interact with the user, the data provided by devices should be analyzed and displayed through graphics or diagrams. The IoT platform should also provide additional tools like platform ecosystems apps, enabling the devices visualization, management and controlling, as well as programming interfaces and software development kits (SDK).

Some of the most used platforms nowadays are Amazon AWS IoT[1], Microsoft Azure IoT[2], Google Cloud IoT[3], Cisco IoT Services[4], Watson IoT (IBM) [5], ThingWorx (PTC) [6], amongst others.

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com
[3] https://cloud.google.com/
[4] https://www.cisco.com/c/en/us/solutions/internet-of-things/overview.html
[5] https://www.ibm.com/watson
[6] https://www.ptc.com/en/products/iiot/thingworx-platform

2.1.2. IoT Communication Protocols

IoT uses lightweight communication protocols in order to produce a more efficient communication in sensor networks. The main advantages are the following: small-size packets, less bandwidth, demand a low demand of processing resources and decrease of energy consumption.

There are different communication protocols that can be used at diverse layers. However, most of these protocols, such as HTTP and TCP/IP, require fast processing units and high-power. This resources and capabilities are not available in embedded devices, being necessary the use of lightweight communication protocols [18]. Thus, application-layer protocols are preferable in order to satisfy the specifications of IoT devices as well as integrate a solution closer to end-users [17]. Three application-layer protocols were used, including MQTT, MQTT-SN and CoAP.

2.1.2.1. MQTT and MQTT-SN Protocols

Message Queue Telemetry Transport (MQTT) was released by IBM and designed for machine-to-machine (M2M) communication in IoT networks. It is a lightweight protocol that has in consideration the latency, bandwidth and the devices with constrained resources.

MQTT is an asynchronous publish/subscribe protocol requiring a low amount of computational resources and network bandwidth. It has a broker that contains topics, where clients publish messages over a specific topic and subscribers receive them after every new update.

MQTT has three QoS (Quality of Service) levels, which defines the reliability in how a message is received [17]:

- Level 1: Fire and Forget – A message is sent only once and no acknowledgment is required. This is a best-effort delivery service.
- Level 2: Delivered at least once – A message is sent at least once and an acknowledgment is required. This guarantees that a message is delivered at least one time to its destination.
- Level 3: Delivered exactly once – A handshake mechanism is used to ensure that only one message is delivered to its destination.

MQTT uses TCP/IP which, in some cases, can be a heavy transport protocol for embedded devices with low processing capacity and battery dependency.

MQTT for Sensor Networks (MQTT-SN) is a protocol similar to MQTT but more focused on the characteristics of sensor networks. It is optimized for implementation on low-cost, battery-operated devices with limited processing and storage capabilities and, unlike MQTT, the MQTT-SN operates on any transport layer and has a smaller packet length to assure the resistance to transmission errors.

Instead of a topic name, that can have until 65535 bytes of size, the MQTT-SN uses topic ID with two-byte long with the intention of reducing the packet's header [19].

In the architecture of MQTT-SN, represented in Figure 3, there are three kinds of components:

- MQTT-SN clients – connect themselves to a MQTT broker via a MQTT-SN GW;
- MQTT-SN gateways (GW) – may or may not be integrated within a MQTT broker. The main function of MQTT-SN Gateway is the translation of messages between MQTT-SN clients and the MQTT Broker;
- MQTT-SN forwarders – MQTT-SN clients can access a GW via a MQTT-SN forwarder in the scenario where the GW is not directly connected to the MQTT-SN clients. The MQTT-SN forwarder encapsulates the MQTT-SN frames received on the wireless side and forwards the received messages unaltered to the GW.

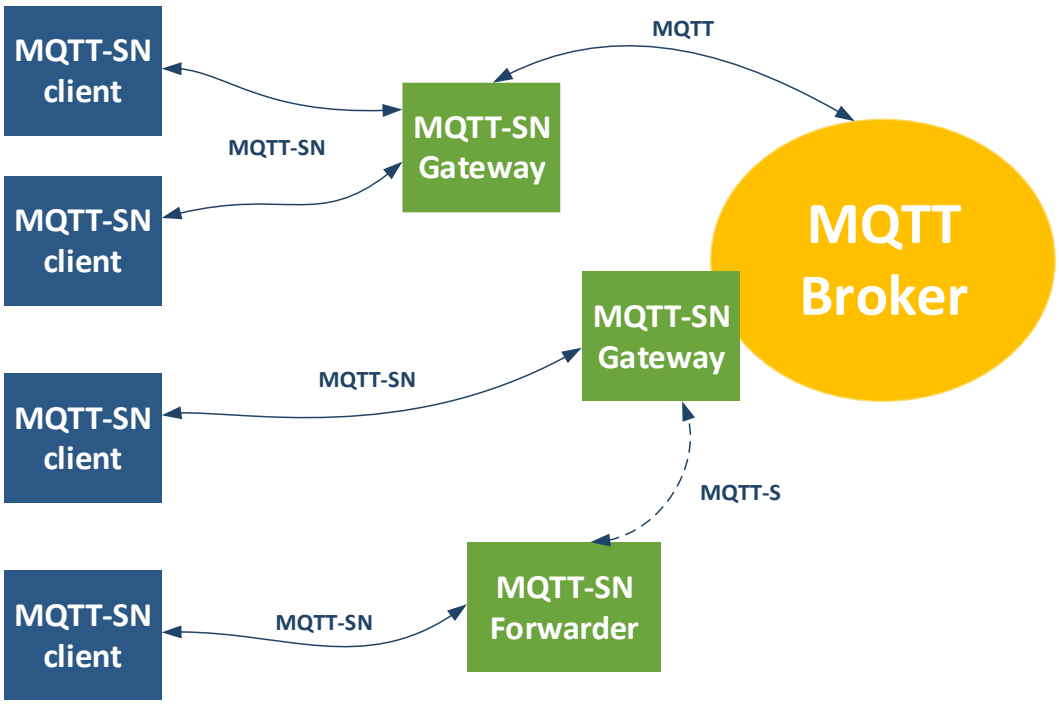

Figure 3 - MQTT-SN Architecture, based on [20]

Depending on how a GW performs the protocol translation between MQTT-SN clients and the MQTT broker, one can enumerate two GW types, as shown in Figure 4. The first type is designated by transparent GW and is responsible to maintain a MQTT connection between the MQTT-SN client and the server. Thus, the GW has the same number of connections on both sides. The second GW type is designated by the aggregating GW, which has only one MQTT connection to the server. In this case the messages are exchanged amongst each MQTT-client and the aggregated GW via a specific connection. Then, the GW decides the correct order for the messages to send through a single connection to the server [20].

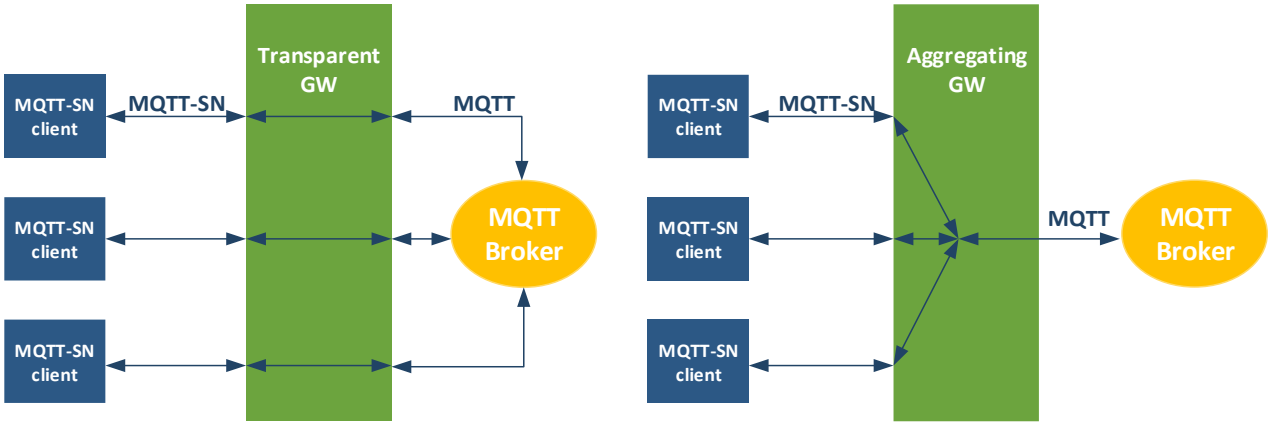

Figure 4 - Transparent and Aggregating Gateways, based on [20]

2.1.2.2. CoAP

Constrained Application Protocol (CoAP) (RFC 7252) was designed by the Internet Engineering Task Force (IETF) for peripheral devices with limited resources. It is a request/response protocol essentially used for machine-to-machine (M2M) communication. It runs over UDP and is interoperable with HTTP, using HTTP commands to allow the communication between client and server. This communication also requires a gateway to adjust the transport logical connection of TCP (HTTP side) with the transport connectionless session of UDP (COAP side). In addition, the CoAP uses Universal Resource Identifier (URI) instead of topics as used by MQTT.

Considering the fact that CoAP does not depend on a reliable TCP connection, this implies that COAP can operate in scenarios with low bandwidth and poor signal situations.

To achieve reliability, CoAP has two bits reserved in the header to indicate the type of message and the QoS level [14]:

- Confirmable – a request message that requires an acknowledgement as a confirmation of its reception
- Non-Confirmable – a message that doesn't need an acknowledgement
- Acknowledgement – it confirms the reception of a confirmable request message
- Reset – it confirms the reception of a message that couldn't be processed

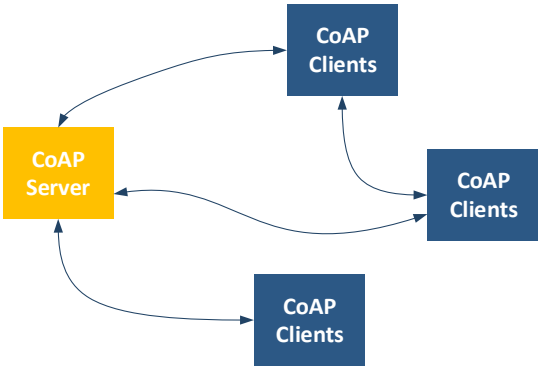

Figure 5 - CoAP

2.1.2.3. Comparison MQTT, MQTT-SN and CoAP

Hamdani et al. [17] compared the three MQTT QoS (Quality of Service) levels with CoAP, regarding message delivery latency and network bandwidth usage. In terms of latency, the MQTT shows more stability and the CoAP protocol evidences lower rates, whereas in terms of bandwidth the MQTT protocol consumes more than the CoAP does. These results are due to the QoS of MQTT since CoAP messages require fewer packet bytes while ensuring acknowledgement of the received messages.

Karagiannis et al. [14] evaluated the performance of CoAP and MQTT at different conditions using a common middleware that allows a more suitable protocol to be used in specific network conditions. MQTT messages experienced lower delays than CoAP for low packet loss and higher delays for higher packet loss. When messages have a small size and low loss rate, CoAP generates less extra traffic than MQTT to ensure the transmission reliability of data messages.

Silva et al. [19] compared the MQTT and MQTT-SN protocol regarding memory consumption by messages header. Considering only the header size plus a payload of the collected temperature, MQTT-SN presented a better performance when compared to the MQTT in approximately 50% due to its shorter packet's header. The fact that MQTT-SN uses a topic ID instead of a string helps decreasing the messages' size.

Thus, CoAP and MQTT-SN can be considered as good options in situations with a low level of available bandwidth. However, in scenarios where the bandwidth is not an issue and reliability is needed, the MQTT is the best solution.

Table 1 details the comparison between the three IoT application-layer protocols discussed in this sub-section.

Table 1 – Comparison between MQTT, MQTT-SN and CoAP [21]

	MQTT	MQTT-SN	CoAP
Year	1999	2007	2010
Architecture	Client/Broker	Client/Gateway Client/Forwarder Gateway/Broker	Client/Server or Client/Broker
Abstraction	Publish/Subscribe	Publish/Subscribe	Request/Response or Publish/Subscribe
Header Size	2 bytes	2 bytes	4 bytes
Message Size	Small and Undefined (256 MB maximum size)	Maximum size 65535 bytes	Small and Undefined (normally small to fit in single IP datagram)
Semantics / Methods	Connect, Disconnect, Publish, Subscribe, Unsubscribe, Close	Defines 27 type of control message, such as Connect, Publish, Unsubscribe, Disconnect	Get, Post, Put, Delete
Quality of Service (QoS)	QoS 0 – At most once (fire-and-forget) QoS 1 – At least once QoS 2 – Exactly once	QoS -1 (within Publish messages sent by a client) QoS 0 – At most once QoS 1 – At least once QoS 2 – Exactly once	Confirmable Message Non-confirmable Message
Standards	OASIS, Eclipse Foundations	MQTT-SN is not standardized. It is a copyright of IBM	IETF, Eclipse Foundation
Transport Protocol	TCP	UDP	UDP, SCTP
Security	TLS/SSL	DTLS	DTLS, IPSec
Licensing Model	Open Source	Open Source	Open Source
Organizational Support	IBM, Facebook, Eurotech, Cisco, Red Hat, Software AG, Tibco, ITSO, M2Mi, Amazon AWS, InduSoft, Fiorano	IBM	Large Web Community Support, Cisco, Contiki, Erika, IoTivity

2.2. Virtualization

The exponential increase of IoT devices connected to the Internet leads to security weaknesses in local edge networks. Furthermore, the communication between these local devices and the remote cloud incurs in high delays and a high consumption of bandwidth resources. An approach to enhance the mitigation of these issues introduces by IoT devices is to use virtualization layers [22]. A virtualization layer allows better automation and the possibility to segment units in the network, making it possible to separate, for example, the control layer from the network topology.

Using virtualization breaks the connection between a service or application and the network topology, creating a network resource abstraction, encouraging application development and its easier deployment. In addition, virtualization isolates, supports configuration and management of virtual instances, allowing network resources to share common features in a more efficient and flexible way [23].

There are three types of virtualization: hardware virtualization, operating system – level virtualization and network virtualization. These diverse types of virtualization are discussed in the next sub-sections.

2.2.1. Hardware Virtualization

The hardware virtualization consists in running multiple and independent virtual machines (VMs). Each of these VMs operates as an emulated standalone physical machine. Hence, it is necessary to use a software that creates and deploys the VMs, each one with their own computing and networking devices, operating system (guest OS), and their own applications. Using this type of virtualization, it's possible to eliminate some eventual incompatibilities between applications and operating systems as well as to take advantage of the full capacity of the computer, optimizing the use of its physical resources, space and associated cost.

The hypervisor is the software already mentioned in the previous paragraph. It provides an abstraction of hardware resources and decides how they should be virtualized, offering to each VM its own processing and networking resources. There are two types of hypervisors. The type-I hypervisors don't need an operating system, running directly on top of the hardware. This allows them to be more efficient, scalable and with a convenient performance. Some examples of type-I hypervisors are Hyper-V, vSphere[1] and ESXi[2] developed by VMWare and XenServers[3]. The type-II hypervisors run upon the physical host operating system, having the possibility to use more configurable hardware. Another advantage of this type of hypervisor is that it can be installed on a regular desktop system. Examples of type-II hypervisors include VMWare Player[4] and VirtualBox[5].

[1] https://www.spherestandards.org/
[2] https://www.vmware.com/products/esxi-and-esx.html
[3] https://xenserver.org
[4] https://www.vmware.com/products/workstation-player.html
[5] https://www.virtualbox.org/

Although the hypervisor allows running multiple and different operating systems at the same time, it demands for a suitable amount of physical computing resources such as CPU and memory. In the case the physical resources available are not enough for the entire set of VMs, the complete system could have a very low level of computing performance [23].

2.2.2. Operating System – Level Virtualization

The OS-level virtualization provides containers that are lightweight operating systems isolated from each other but sharing the kernel resources provided by the physical host. This fact limits the flexibility of the containers but allows a better virtualization performance when compared to the virtual machines. Sharing the kernel makes disk images used by containers smaller than the ones used by hypervisors that run full operating systems in VMs [24]. Thus, containers remove the overheads associated to the guest OS, offering better performance than virtual machines. Another advantage of containers when compared to VM's is the lower booting up time as they use the kernel of the physical host [23].

Table 2 and Figure 6 show the main differences between containers and VMs, regarding their architecture and features.

Table 2 – Containers and Virtual Machines features comparison [25][26]

Parameter	Containers	Hypervisor-Based VM
Choice of Operating System	Variant of host OS, as it shares the same kernel	Any
Startup Time	Containers can be booted up in a few seconds	VMs take a few minutes to boot up
Hardware Abstraction and Device Emulation	No	Yes
Disk Image	Small	Large
Storage	Containers take lower amount of storage as the base OS is shared	VMs take much more storage as the whole OS kernel and its associated programs have to be installed and run
Density	High	Low
Security	Not mature and complex	Mature Security models depending on the implementation of Hypervisor

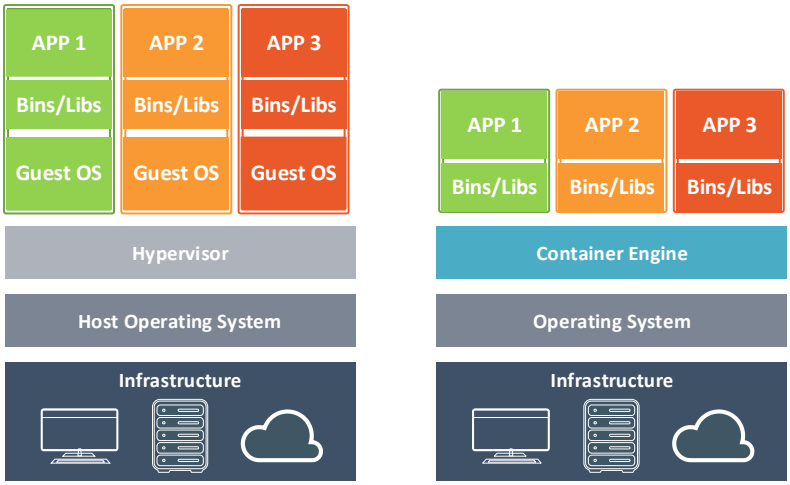

Figure 6 - Containers vs Virtual Machines, based on [27]

Linux Container

The Linux Container (LXC) is an open source container platform, which provides a lightweight virtualization environment that can be used on Linux-based systems.

LXC uses Linux namespaces allowing the creation of multiple isolated Linux virtual environments with its own network devices and a restricted access to file systems. Hence, each container runs independently of each other and it does not get compromised by processes running inside another container. LXC also uses Control Groups (cgroups) to manage and limit the resources consumed by its processes such as CPU, memory, disk I/O [28]. Figure 7 details the LXC architecture.

Using LXC enhances the creation of a virtual environment as close as a standard Linux installation, with the advantage of avoiding a separate kernel. Hence, Linux users can easily create and manage systems or application containers [29]. Some of the images available by LXC are Ubuntu[1], CentOs[2], Fedora[3], Debian[4], amongst others.

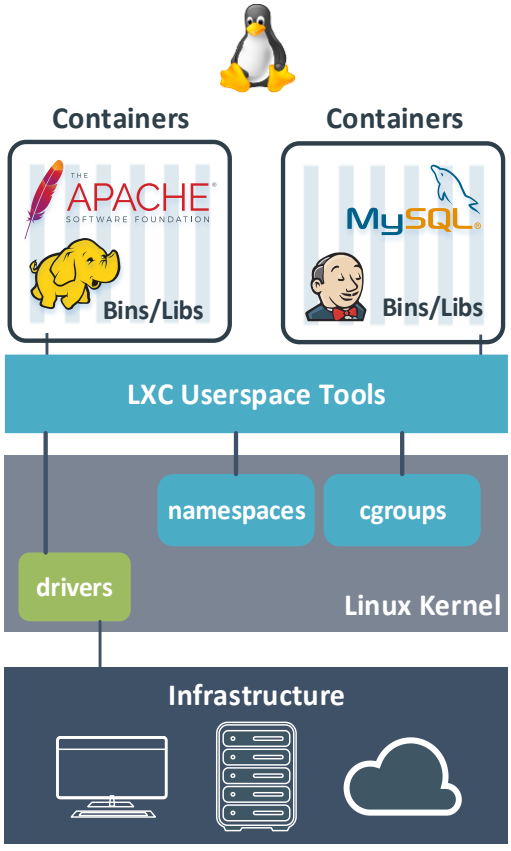

Figure 7 - LXC Architecture

[1] https://ubuntu.com/
[2] https://www.centos.org/
[3] https://getfedora.org/
[4] https://www.debian.org/index.pt.html

Docker

Docker extends from LXC and implements an isolated and protected environment to run an application designated as a container. One of its advantages is the possibility to start with a simple image and add the features needed and desirable to run the application, making it not just an easy and faster way to deploy IoT applications but also a good solution to solve the heterogeneity amongst IoT devices.

The Docker containers help developing, distributing, testing and deploying applications, being these containers managed by the Docker platform.

The Docker architecture is represented in Figure 8. A Docker container is created by a Docker image, which is the basis of a Docker container. This container is the unit where the application services resides and is created and executed by the Docker engine/daemon that manages not only containers, but also images, networks and volumes. The images are built manually or automatically by reading the commands from a Docker file and each given command forms a new layer on top of the previous one. With Docker Cloud and Docker Datacenter is possible to register Docker images and to manage Docker hosts, which can be run on different cloud platforms, thanks to the use of Docker Machine [30] [31].

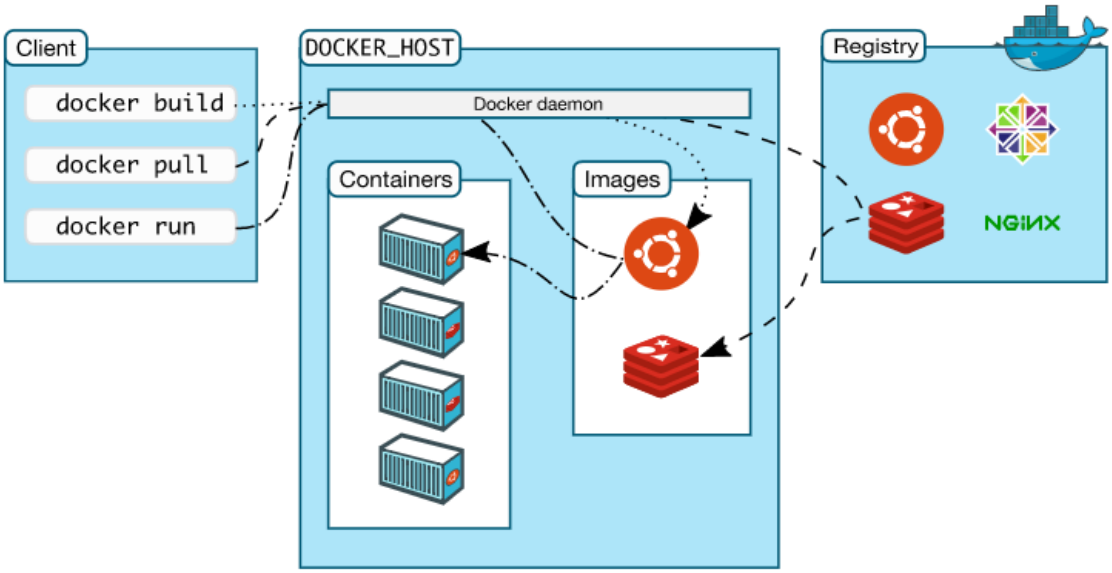

Figure 8 - Docker Architecture [31]

The container technology is in expansion and it's possible to find a large set of choices, *e.g.* OpenVZ[1], Linux Container[2], Docker[3] and Rocket (CoreOs rkt) [4].

Table 3 compares the different features between the mentioned containers.

Table 3 – Container Implementation Comparison [25]

Parameter	OpenVZ	LXC	Docker	CoreOS rkt
Year	2005	2008	2013	2014
Latest Version Release	25th July 2016	LXC 3.2.1 24th July 2019	Version 19.03.3 08th October 2019	Version 2275.1.0 09th October 2019
App or Full-System Container	Full-System	Full-System	App	App
Supported Platforms	Linux (only Virtuozzo Linux)	Linux	Linux, Windows, macOS, Microsoft Azure, AWS	Linux, Windows, macOS
Process Isolation	Uses pid namespace	Uses pid namespace	Uses pid namespace	Uses pid namespace
Resource Isolation	Uses CGroups	Uses CGroups	Uses CGroups	Uses CGroups
Network Isolation	Uses net namespace	Uses net namespace	Uses net namespace	Uses net namespace
Filesystem Isolation	Using chroot	Using chroot	Using chroot	Using chroot
Container Lifecycle	Uses vzctl to manage container lifecycle	Tools lxc-create, lxc-stop, lx-start to create, start, stop a container	Uses Docker daemon and a client to manage the container	Container build tool based on shell scripting, leveraging familiar unix tools

Both OpenVz and Linux Container are full operating system containers, allowing to run multiple applications in a single container but, unlike OpenVZ, LXC has updated release versions as well as an active community around it. Docker and rkt are optimized for application containers, running only individual processes per container.

All the container technologies create a virtualized isolated process and use similar kernel features like namespaces and cgroups. However, LXC allows a better environment for developers, as it authorizes multiple services to run inside a container. This last feature is not possible for Docker or rkt, making them heavier technologies, in terms of physical resources consumption, when compared to LXC.

[1] https://openvz.org/
[2] https://linuxcontainers.org/
[3] https://www.docker.com/
[4] https://coreos.com/rkt/

2.2.3. Network Sensor Function Virtualization

According to IoT Analytics [32], in 2018 the number of connected devices exceeded 17 billion, 7 billion of which were IoT devices. The number of active IoT devices is expected to grow to 10 billion by 2020, and therefore the software abstraction of these devices becomes crucial as well as implementing virtualized functions in order to distribute intelligence in a more optimized way.

The Network Functions Virtualization (NFV) is an initiative of the European Telecommunications Standards Institute (ETSI) and allows the separation of the software from the hardware, enabling the evolution of both in an independent way (Figure 9). With the NFV is possible to create and manage network services through network function abstraction. The NFV enables generic servers to achieve scaling goals, such as the capability of dynamic scaling. To do so, NFV allows network functions to be developed and deployed as software instances running on standard servers [23].

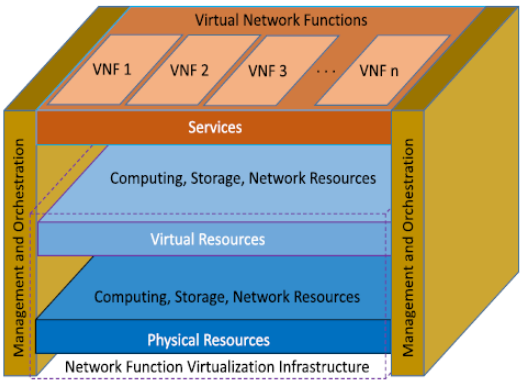

Figure 9 – ETSI Network Function Virtualization Architectural Framework [33]

Considering the scalability, reliability and implementation problems of physical sensors, the sensor virtualization allows the access to the physical properties of physical measurement sensors through virtual sensors [34]. The virtual sensors can aggregate the tasks executed by the diverse physical sensors *e.g.* sensing, processing, communicating and storing. Thus, the virtualization of sensors allows a more abstracted, scalable and flexible interaction between the physical sensors and applications [35].

One of the approaches proposed by JeongGil Ko et al. [36] consists in the introduction of the Sensor Virtualization Module, which solves the limitation problem of applications that use IoT resources. Thus, a software abstraction is provided through open APIs, allowing better data management given by the diverse proposed objects.

Through the SVM engine is made a discovery of IoT devices profiles in the local network, being the characteristics of each device stored in object handles that can be then advertised to the applications. This helps applications selecting the IoT devices with which they want to interact. Hence, SVM enhances the operation of IoT network because it not only simplifies the development of APIs that use IoT devices, but it also exposes these devices that have been found in the local network to the cloud, allowing their management remotely.

Figure 10 shows a scenario where the SVM running on a mobile device manages its local IoT devices and exposes them to the cloud through the SVM application server. Thus, the IoT devices as well as the virtual IoT devices can be accessed by remote applications running on different Internet platforms.

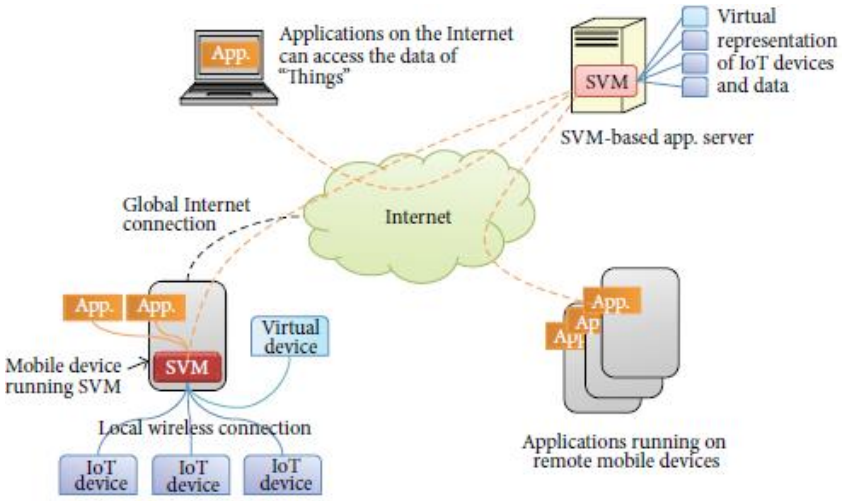

Figure 10 - Sensor virtualization module architecture and usage scenario [36]

The SVM also allows sensor data mash-up. Using this approach, it is possible to combine data from different IoT devices and create a new data group, as shown in Figure 11. Thus, the user doesn't need to connect directly to the physical sensor or retrieve data individually from each device.

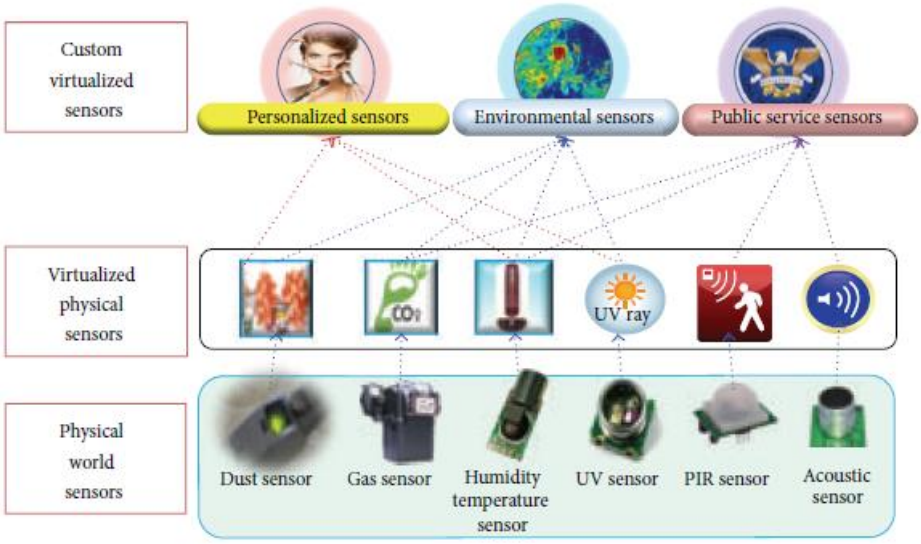

Figure 11 - Sensor data mash-up [36]

Combining both network functions and virtualized sensors ensures running services strictly when necessary, such as, when a client needs to connect to sensors in order to collect data or just start and stop services. Without using virtualization, the access and management of the data and resources provided by IoT devices could be severely restricted due to the manufacturer's software.

2.3. Fog and Edge Computing

Fog computing was proposed in 2012 by Cisco and is a geographically distributed computing architecture, where different devices at the edge of the network are connected to provide computation, networking, decision making, data management and storage services. It is an extension of the cloud service to the edge of the network and it is located closer to the end-users in order to provide them location-based services with minimum access latency.

It is composed by fog nodes, which includes network edge devices, like routers, gateways, switches, etc., and management systems within these devices. These fog nodes are distributed between the devices and the cloud, where they might be static or mobile, which is an important fact in mobile scenarios like smartphones or vehicles. They store temporarily the received data generated by the devices, and process some of those data, transferring the rest to the cloud. Hence, there is a reduction on the data volume exchange through the network infrastructure, helping to reduce the latency.

The fog architecture, represented in Figure 12 is composed by the terminal layer, containing the different IoT devices and sensors responsible to receive data and transmit this to the upper layer. In addition, there is the fog layer, which is located at the network edge and is formed by diverse fog nodes. Adding this extra layer between the IoT devices and the cloud increases the performance, mobility and security, by encrypting and isolating data. At least, there is the cloud layer, which contains high-performance servers and storage devices and provides application services [3].

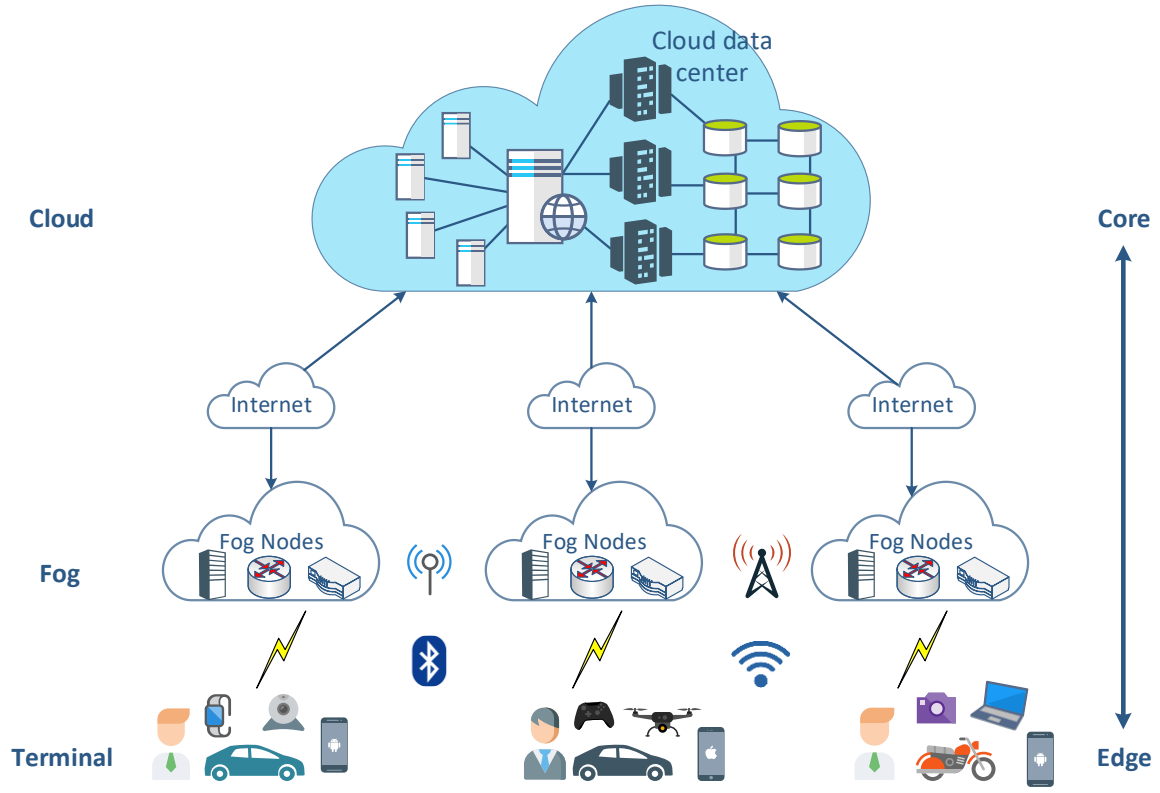

Figure 12 – Hierarchical architecture of fog computing, based on [3]

The communication between the devices and the fog nodes, including the intercommunication between these nodes, is made essentially by wireless access technologies. The nodes are linked to the cloud by IP core network.

Edge computing is a computing model that also extends cloud services to the edge mobile devices. This model is also commonly designated as multi-access edge computing [37]. It enhances management, storage and the processing power of data generates by connected devices.

As seen in Figure 13, the edge computing architecture is composed by edge nodes and devices, like smart sensors, smartphones, etc., that make decisions, process and storage data when they have the capacity to do so. The edge nodes can create local edge networks or communicate with the cloud through the core network. In this model, when end devices need services, they request them from the cloud. In addition, the end devices can provide services to other requesting devices.

Edge computing helps reducing the latency and the amount of traffic between the mobile devices and the cloud. It also provides services, security and privacy protection [3].

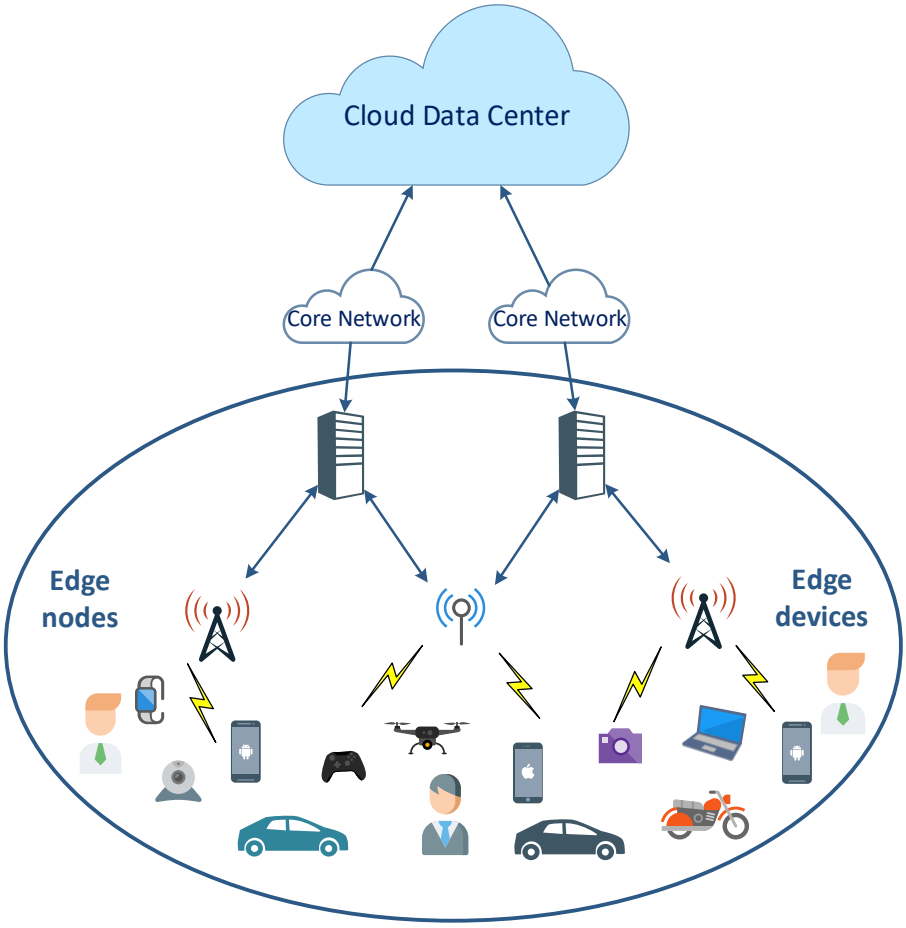

Figure 13 – The architecture of edge computing, based on [3]

Puliafito et al. [38] presented a survey on how fog computing can support IoT devices and services. They discuss the main outcomes offered by fog computing, including six IoT application domains that benefit from using this paradigm. Fog computing extends the cloud towards the network edge, where fog services can be distributed anywhere along the cloud-of-things continuum. Depending on its requirements, a fog service may be resource-rich end devices (*e.g.* video surveillance cameras), advanced edge nodes (*e.g.* switches, gateways) and specialized core networks routers.

A. Yousefpour et al. [39] compared fog computing with other related computing paradigms such as edge computing (see Figure 14). It mentions the OpenFog Consortium [40] that considers fog computing as "a horizontal, system-level architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing continuum", whereas edge computing is more likely to be limited to processing at the edge. Thus, fog computing is considered a more general and comprehensive definition scope, being suitable for many cases in the IoT environment. This article also provides a taxonomy of research topics in fog computing.

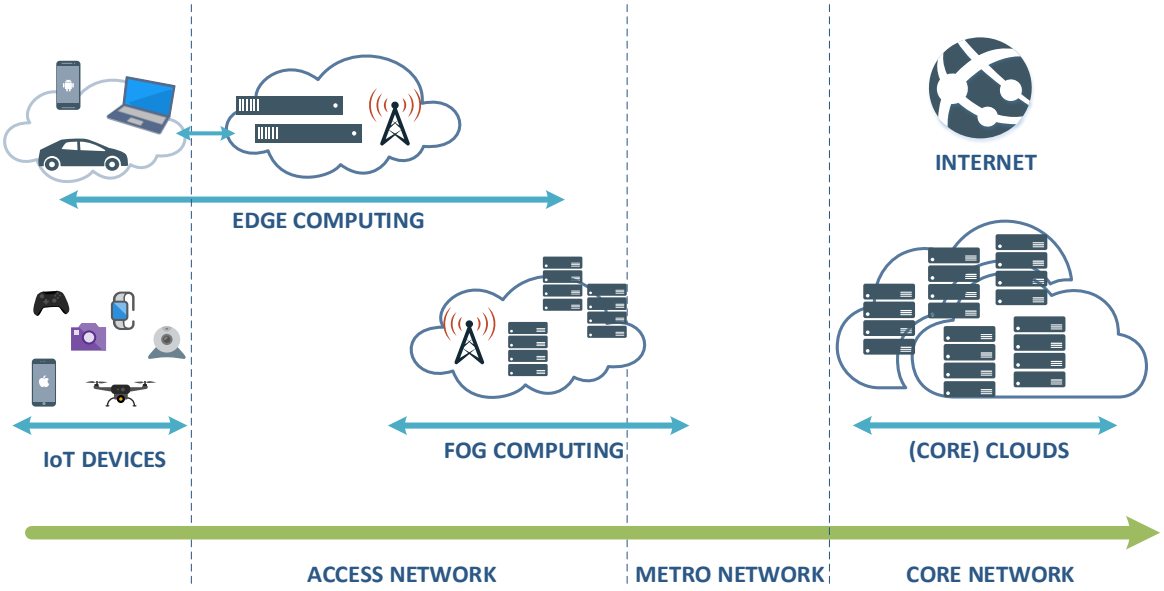

Figure 14 – Comparison of fog computing and edge computing, based on [39]

Table 4 lists some relevant similarities and differences between fog and edge computing.

Table 4 – Similarities and differences between fog and edge computing [3]

	Fog Computing	Edge Computing
Architecture	Hierarchical, decentralized, distributed	Hierarchical, decentralized, distributed
Proximity to end devices	Near (single network hop or few network hops)	Located in end devices
Latency	Low	Low
Bandwidth costs	Low	Low
Resource	Limited	More limited
Computation and storage capabilities	Limited	More limited
Mobility	Supported	Supported
Scalability	High	High
Energy consumption	High	Low
Service	Virtualization	Virtualization
Location of data collection, processing, storage	Near-edge and core networking, network edge devices and core networking devices	Network edge, edge devices
Handling multiple IoT applications	Supported	Unsupported
Resource contention	Slight	Serious
Focus	Infrastructures level	Things level

2.4. Software-Defined Networking

The Software-Defined Networking (SDN) is a network architecture that enables the creation of independent features and protocols of manufacturers, overcoming the problems related to the closed hardware and proprietary software [41].

2.4.1. SDN Architecture

The SDN allows a separation of the control plane from the data plane. The data plane is where the traffic flows are physically exchanged. It is composed by network elements, such as routers and switches, that expose their capabilities and resource status to the control plane.

This architecture has a centralized control commanded by the SDN controller. The SDN system can also manage the network resources through SDN applications. The SDN controller has an overview of the network topology as well as the network resource status, allowing it to improve and adapt the packet routing rules as well as face the challenges of scalability, performance and security. This way, the network devices in the data plane take actions by following the rules established by the controller [41].

The controller also translates the SDN applications policies to low-level control instructions, and therefore they can be executed by the network devices on the data plane. It also enables the programmability in the network through an Application Programing Interfaces (API) that allows the SDN applications to inform the controller about their network service requests.

Thus, the SDN can be organized into three planes: the data plane, control plane and application plane, as can be seen in Figure 15. The SDN typically has two interfaces. The first interface exists between data and control planes and is designated as the Southbound API. The second interface exists between each application and the SDN controller and is designated as the Northbound API [23].

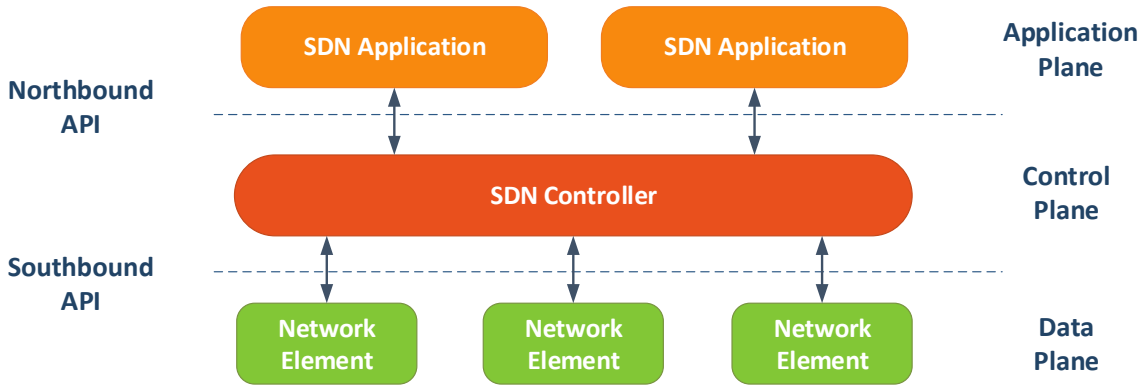

Figure 15 - General architectural framework for SDN, based on [23]

2.4.1.1. Network Elements

The lowest layer of SDN architecture is composed by the data plane that communicates with the SDN controller through the Southbound API, enabling the elements to update the controller about their capabilities and status of their resources. Therefore, the controller can have a global view of the network and thus send this information to the application plane. The controller can also instructs the data plane by configuring rules in the network elements [42].

The most used Southbound API protocol is the OpenFlow. It was released in December 2009 and is managed by the Open Networking Foundation (ONF) [43].

The OpenFlow is a standard protocol that allows the communication between the SDN controller and the network infrastructure devices, whether they are physical or software-based. It is used by the SDN controller to discover the network topology and modify data flow rules in the flow tables of switches [44].

There are two types of OpenFlow switches. The pure switches only support OpenFlow and have no legacy features or on-board control, depending completely on the SDN controller for performing decisions about how to forward the traffic through the data plane. The hybrid switches support the

OpenFlow protocol as well as other legacy algorithms or networking protocols [45]. The Open vSwitch is an example of a hybrid software-based switch. The main components of an OpenFlow switch are represented in Figure 16.

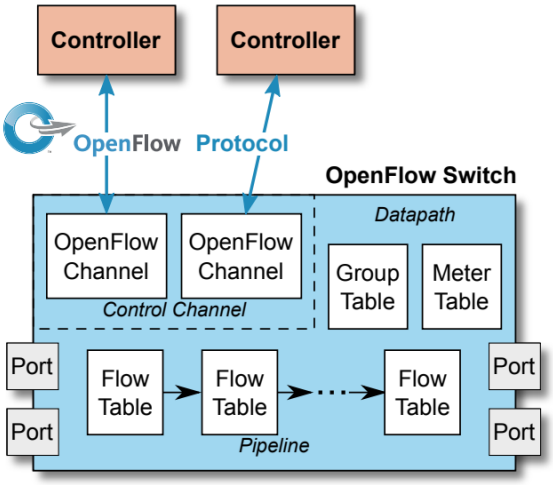

Figure 16 - Main components of an OpenFlow switch [46]

An OpenFlow switch contains at least three parts [44]:

- A pipeline of flow tables, which performs packet lookups and execute a set of instructions or actions that determine how to handle the packets;
- Secure Channel, that enables the communication between the switch and the controller, allowing the packets flow and instructions;
- The client side of the OpenFlow Protocol.

Open vSwitch

Open vSwitch (OVS) is managed by The Linux Foundation and is a multilayer software licensed under the open source Apache 2 license. It is the most popular virtual switch implementation and it can be used as virtual switch in virtualized environments and as a general software switch.

The major part of the code is written in platform-independent C and is easily ported to other environments. OVS supports common standards protocols such as OpenFlow, LACP, etc [47][48].

The support of the OpenFlow protocol makes Open vSwitch a great addition to any SDN-based network.

OpenFlow Messages

The OpenFlow protocol has three types of messages [45]:

- **Controller-to-switch**, that are initiated by the controller to set and query configuration parameters in the switch, to collect statistics and manage the state of the switch, being used to add /delete or modify flow table entries, and to send packets out of a determined port.
- **Symmetric**, that are initiated by the controller or the switch, like Hello or Echo messages to discover the latency, bandwidth and heartbeat.
- **Asynchronous**, that are initiated by the switch to inform the controller about switch state changes, packet arrival and error messages. When an idle timeout value is indicated, the entry should be removed according to that value and / or a flow removal message should be sent to the controller.

A Packet-In message is asynchronous, being sent from the switch to the controller for further processing. Wherever there is no matching flow entry and there is a default entry with a *send to controller* action, a Packet-In message is sent to the controller. This message transports a copy of the received message in the switch.

When the controller receives a Packet-In message, it adds an action list field, containing a list of actions that defines how the switch should handle the packet received in the initial Packet-In message. The previous list of actions is sent to the switch, using a Packet-Out message. If the controller does not know the localization where the destination host of the initial received packet is attached to the network topology, the controller sends to the switch a Packet-Out to perform flooding.

The Modify-State messages, also called ‘flow mod’, are messages sent by the controller to manage the state of the switches as well as add, delete or modify flow table entries.

OpenFlow Controller Behavior

There are two types of controller behavior: the reactive mode and the proactive mode.

In the reactive approach, the first flow packet received by the switch triggers the controller to insert flow entries in each OpenFlow switch of the network. Despite being an efficient use of the flow table memory, every time a new flow is introduced, it adds an additional setup delay in the system operation, due to the Round-Trip Time between the switch and the SDN controller. Another disadvantage is the dependency on the controller which in case of a connection loss between the switch and the controller forces the switch to drop flow packets with unknown local rules to apply to.

In the proactive behavior, the controller previously fills the flow table in each switch, avoiding the additional flow setup time of the reactive mode. Having the forward rules in the flow table makes the switch less dependent on the controller, having eventually no traffic disruption in the case of a temporarily connection loss between the controller and the switch [49].

2.4.1.1.1. Controlling Example

Figure 17 shows an example of SDN controlling. In SDN, when the switch receives the first packet from a new flow (step 1), it checks if there is a forwarding rule matching some of the header fields of that received message in the local OpenFlow table (step 2). If it finds a match entry, the instruction (*i.e.* action) is executed, being the packet forwarded to the receiver (step 5). However, if there is no match in the flow table, a flow request (*packet_in* message) is sent to the controller (step 3) asking in which action should be taken. The controller then responds with a flow-entry that contains the switch output port from where the packet should be forwarded (step 4). The switch adds, delete or update the flow table depending on the flow control message sent by the controller. Once it has added a new entry in the switch, it is now capable to send the packet to the receiver (step 5) [5].

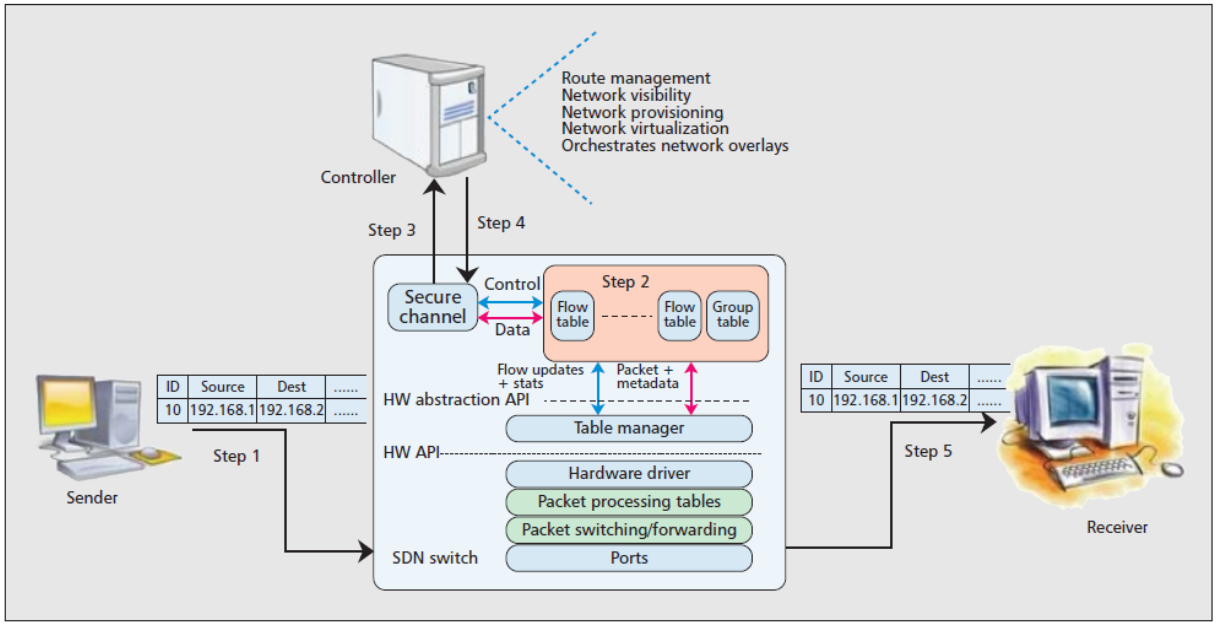

Figure 17 - The operation of SDN (controller - switch) [5]

2.4.1.2. SDN Controllers

2.4.1.2.1. OpendayLight

OpenDayLight (ODL) is an open source SDN project created in 2013 by the Linux Foundations with nearly 50 major corporate members, including Cisco, HP, IBM, Juniper and VMWare. These companies have contributed with the necessary resources to support the creation of an SDN platform. In addition, ODL is available under the Eclipse Public License (EPL), increasing the compatibility with the expansive environment of libraries and third-party components. Thus, ODL is a modular open source platform for SDN, implemented in Java programming language and operated through an open and active community [50].

OpenDayLight was the first controller entering the IoT domain with the Lithium release in June of 2015. The IoT Data Management (IoTDM) module was implemented, enabling authorized applications to retrieve IoT data uploaded by any device. In addition, the IoTDM also acts as a oneM2M compliant, which is the global standardization body for IoT that prepares, approves and maintains the necessary functions of IoT technologies.

OpenDayLight supports a layered architecture. The Network Applications & Orchestrations layer on the top consists on business and network logic applications that use the controller to implement self-adaptive management actions over the network infrastructure. These applications also run algorithms to perform data analytics in order to orchestrate new and more efficient control rules to manage the resources of the network infrastructure. The middle layer, the Controller Platform, is the framework which enables SDN abstraction. The controller exposes northbound APIs to the application layer, using the Open Services Gateway Initiative (OSGI) framework and bidirectional REST API. To implement protocols for command and control of the physical hardware within the network, the controller uses the southbound API, that supports multiple protocols like OpenFlow, NETCONF, BGP-LS, SNMP, LISP, etc. The bottom layer consists of the physical and virtual devices, such as switches, routers, open vSwitches etc [51].

OpenDaylight is one of the most featured controllers, being able to be used in any operating system, claiming extensive compatibility support from vendors, presenting a good GUI feature and documentation. [52]

2.4.1.2.2. RYU

Ryu means “flow” in Japanese and is a component-based and open source software defined by a network framework. It is written completely on Python, and provides well-defined Application Program Interfaces (APIs), allowing developers to create new network management and control applications in an easy way. All the code is available under the Apache 2.0 license and can be found on GitHub, which is provided and managed by the Ryu community. Developers can modify existing code or implement their own from scratch.

Ryu controller supports various southbound protocols for managing network devices, such as Network Configuration Protocol (NETCONF), OpenFlow Management and Configuration Protocol (OF-config), OpenFlow, which is the most popular southbound protocol, up to the latest available OpenFlow version, *i.e.* 1.5 [53].

Ryu applications (python scripts) listen to events that handles the received messages. The app_manager.py is the file that loads the Ryu applications, which can be used individually or in an integrated way [54].

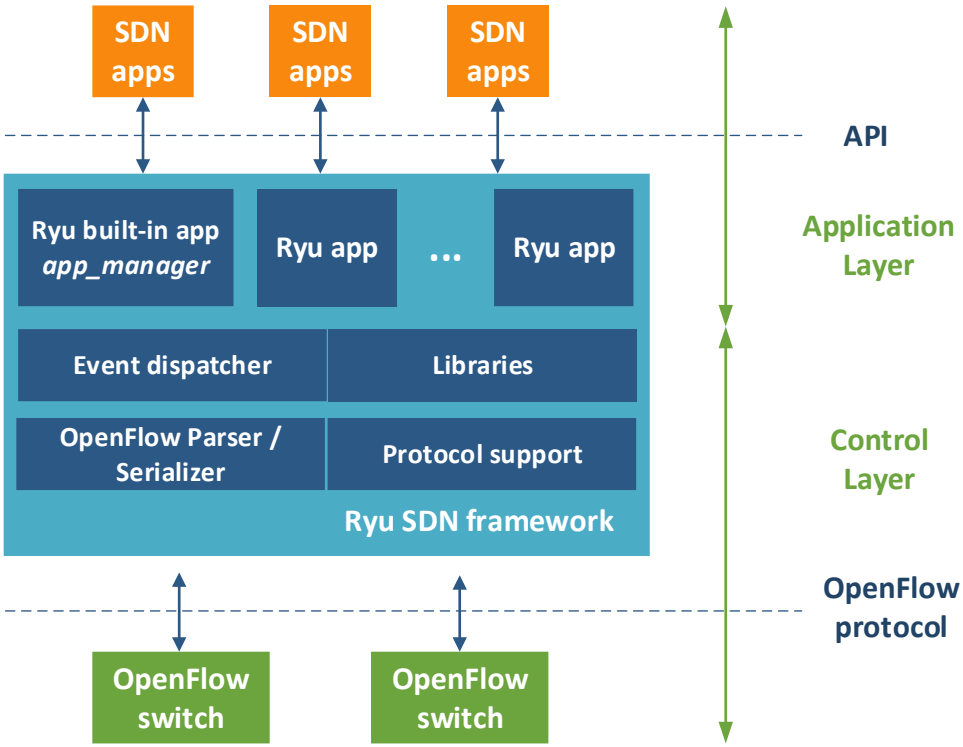

Figure 18 - Ryu Framework, based on [55]

Considering the fact that Ryu has a fair number of features, uses Python and it has an exclusive support for Linux OS, makes Ryu a good choice for small scale SDN deployments and research applications [52].

2.4.1.2.3. Comparison Amongst Several SDN Controllers

Recently, there have been several works that contributed to an accurate benchmark of different SDN controllers.

Shalimov et al. [56] performed a comparison regarding throughput and latency, scalability, reliability and security. In terms of security, Ryu was the one who best coped four of the five tests. Erickson et al. [57] mentions the importance of the programing language used by an SDN controller, claiming Java as a good choice as it runs across platforms and it supports multithreading, whereas Python has an inability to support multithreading. The results publicly available in Khondoke et al. [58] show that Ryu was the best controller using a Multi-Criteria Decision Making (MCDM) method called Analytical Hierarchy Process (AHP). In addition, OpenDayLight got the next better results. In [59], OpenDayLight was one of the controllers with good documentation and flexibility. Rowshanrad et al. [60] shows that OpenDayLight, when compared to Floodlight, has the best latency results under low traffic loads. In [52], Salman *et al.* measured latency and throughput performances under a different number of switches and threads, concluding that OpenDayLight was a good choice as a full-featured controller. Mamushiane et al. [42] performed tests based on latency and throughput.

Table 5 presents a brief comparison amongst the diverse SDN controllers that we have just discussed in the current sub-section.

Table 5 - Feature-based comparison of SDN controllers [42]

	RYU	Floodlight	OpenDayLight	ONOS
Southbound API	OF 1.0, 1.2, 1.3, 1.4, 1.5 NETCONF, OFCONFIG, OVSDB	OF 1.0, 1.1, 1.2, 1.3, 1.4, 1.5	OF1.0, 1.3, 1.4, 1.5 NETCONF/YANG, OVSDB, PCEP, BGP/LS, LISP, SNMP	OF1.0, 1.3, 1.4, 1.5 NETCONF
REST API	Yes (For SB only)	Yes	Yes	Yes
GUI	Yes (Initial phase)	Web / Java-based	Web-based	Web-based
Modularity	Medium	Medium	High	High
Orchestrator Support	Yes	Yes	Yes	No
OS Support	Most supported on Linux	Linux, Windows, and MAC	Linux, Windows, and MAC	Linux, Windows, and MAC
Partner	Nippo Telegraph And Telephone Corporation (NTT)	Big Switch Networks	Linux Foundation With Memberships Covering Over 40 Companies, Such as Cisco, IBM, NEC	ON.LAB, At&T, Ciena,Cisco, Ericsson,Fujitsu, Huawei,Intel, Nec,Nsf.Ntt Comunnication, Sk Telecom
Documentation	Medium	Good	Very good	Good
Programming Language	Python	Java	Java	Java
Multi-threading Support	Yes	Yes	Yes	Yes
TLS Support	Yes	Yes	Yes	Yes
Virtualization	Mininet and OVS	Mininet and OVS	Mininet and OVS	Mininet and OVS
Application Domain	Campus	Campus	Data center and Transport-SDN WAN	Data center and Transport-SDN WAN
Distributed / Centralized	Centralized	Centralized	Distributed	Distributed

2.4.1.3. Northbound Management

The Northbound API is used to allow the communication between the SDN controller and the SDN services and applications running at the topmost architecture layer. It provides an abstraction of network functions, enabling applications to program the network and implement functionalities regardless of the underlying layers. The Representational State Transfer (REST) API is currently the most used API by SDN controllers [42].

REST API improves system extensibility by allowing clients to download and execute code after deployment. It uses HTTP requests to post, read and delete data through GET, POST, PUT and DELETE methods. GET request is used to obtain information, *e.g.* get the switch id, POST and PUT request to update an existing resource, *e.g.* get all flow entries, and DELETE to delete resources, *e.g.* delete all flow entries of the switch. In the case of the POST method, its body is used to transfer data structured in JSON or XML [61].

2.4.2. Software-Defined Networking Virtualization

As mentioned in previous chapters, the number of IoT devices is growing continuously, making traditional network infrastructures inadequate to handle all the produced data. Besides, the manual configuration of network devices could lead to errors with the increase of devices [62].

Through the use of SDN and NFV it is possible to have an abstract view of physical network infrastructures as well as an easier operation and maintenance, allowing to reach the necessary scalability and flexibility. Although SDN and NFV are different in terms of concept, architecture and functions, they complement each other, extracting the full potential when they coexist. NFV allows the migration of controllers to optimal locations and the virtualization of physical network devices, such as routers and firewalls. Whereas SDN provides programmable network under a central control, allowing a quicker and dynamic installation of policies in order to enable service functions [23] [63] [64] .

Ojo et al. [65] and L. Valdivieso et al. [66] proposed different architectures with the application of SDN with NFV in order to address the challenges of IoT. These articles are discussed in the next section.

2.5. Related Work

In this section, the relevant literature that relates to the technologies explored in this thesis are discussed.

Table 6 - List of Related Research

Research's Title	SDN	Virtualization	Fog / Edge Comp.	IoT	Advantages / Disadvantages
SDN-based architecture challenging the IoT heterogeneity (I. Bedhief, M. Kassar, and T. Aguili, 2016) [67]	Yes	Yes (Containers)	No	Yes	+ tests the connectivity between containers, managed by an SDN controller - no use case; doesn't consider fog/edge options
Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies (R. Morabito and N. Beijar, 2016) [68]	Yes	Yes (Containers)	Yes (Edge computing)	Yes	+ practical use case working with containers; network management and data processing at the edge
SDN docker: Enabling application auto-docking/undocking in edge switch (Y. Xu, V. Mahendran, and S. Radhakrishnan, 2016) [69]	Yes	Yes (NFV / Containers)	Yes (Edge Switches)	Yes	+ practical use case working with containers; different SDN controller, created specifically for the tested scenario; use of MQTT protocol; docking/undocking capability
A SDN-IoT architecture with NFV implementation (M. Ojo, D. Adami, and S. Giordano, 2016) [65]	Yes	Yes (NFV)	Yes (Edge nodes)	Yes	+ SDN/NFV edge nodes that allow fast deployment of new services - doesn't use containers
SDN/NFV Architecture for IoT Networks (L. Valdivieso, et. al, 2018) [66]	Yes	Yes (NFV)	Yes (Edge Switches)	Yes	+ virtualization layer; orchestration and management layer - doesn't use containers; scenario emulated in mininet
Software-Defined Fog Network Architecture for IoT (S. Tomovic, K. Yoshigoe, I. Maljevic, and I. Radusinovic, 2017) [70]	Yes	Yes (Hypervisor / Containers)	Yes (Fog)	Yes	+ practical use cases; SDN controller modification to orchestrate fog nodes; containers / VMs with IoT services

As can be seen in Table 6, all the researches included SDN as part of the proposed designs/architectures applied to IoT scenarios. SDN is an important network architecture for all the technologies mentioned in this thesis, due to its flexibility, scalability, security and dynamic properties.

Bedhief et al. [67] proposed an architecture based on SDN and Docker techniques to manage devices and networks heterogeneity. This architecture was designed so the IoT devices could use its network interface to communicate through an SDN-based system, which is composed by diverse SDN-based switches and a POX SDN controller. They have tested three scenarios, *i.e.* the communication between two IoT devices, the communication between a device and a docker, and the communication amongst dockers. Observing the TCP and UDP traffic, and despite being possible to establish communication in all three scenarios, the highest delay was recorded between dockers, resulting on the fact that docker needs more control than hosts defined in Mininet. Morabito et al. [68] also proposed a design using container-based virtualization technologies in order to create an IoT gateway that allows the management of different services. These services are included inside a Docker container and to be able to attribute a container to a specific user and ensure its isolation, they used Open vSwitch and an SDN Controller (not mentioned which one) that resides in the IoT platform, *i.e.* at the edge computing. In addition, they have an orchestrator that determines the software used to process the sensors data and if it is more suitable to deploy in the gateway or in a data center. Another approach involving SDN and container-based technology is the one described by Xu et al. [69] which proposed an SDN architecture with an in-house controller to enable auto-docking / undocking of applications at the edge switches. In the study it was tested the connection capability between the SDN docker switch and a remote end-host for application download. Thus, an automatic deployment framework was created for analyzing incoming packets received from the end-user, getting the necessary applications from a central repository that stores the binary images, and then installed or uninstalled in the switch.

In order to help solving the different challenges in IoT environments, other virtualization technologies, other than containers, are used. Ojo et al. [65] presented an SDN-IoT architecture with network function virtualization (NFV). The combination of these two technologies can help solving problems such as interoperability of heterogeneous devices, scalability, discoverability, security, management and application specific requirements. They used SDN/NFV edge nodes to allow the fast deployment of new services allowing them to deliver services such as ultra-low latency and high bandwidth to IoT applications. L. Valdivieso et al. [66] proposed an alternative approach using an architecture that integrates SDN and NFV focusing on IoT environments. As shown in Figure 19, the architecture has four layers as follows : the Infrastructure Layer that includes the hardware and basic software to forward the traffic; the Control and Virtualization Layer, which is composed by the control plane and virtualized elements; the Application Layer, where the different NFV applications are located; and the Orchestration and Management Layer, that manages the other different layers of the infrastructure, which is required when the data and control plane are separated.

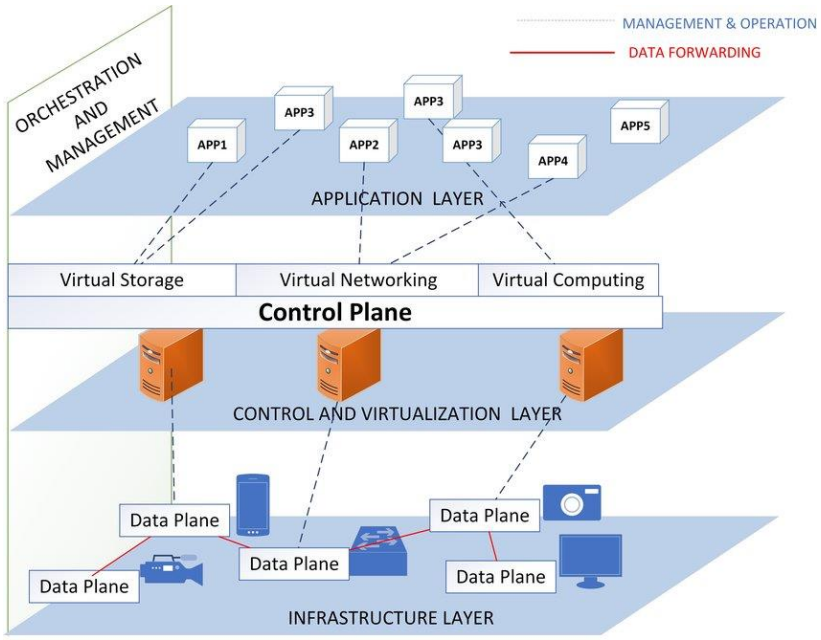

Figure 19 - IoT SDN/NFV Architecture [66]

In previous researches, SDN, virtualization and edge computing were applied to IoT scenarios. Tomovic et al. [70] proposed a system model of IoT architecture taking advantage of SDN and fog computing paradigm in order to support a high level of scalability, real-time data delivery and mobility. The use of SDN allows a fog orchestration delegated to SDN controller. Hence, combination of these two technologies enhances the ability to handle a large volume of fog nodes as SDN controller can delegate tasks to fog nodes. The system structure is represented in Figure 20 and involves end devices, SDN controllers, heterogeneous fog infrastructure and cloud in the network core.

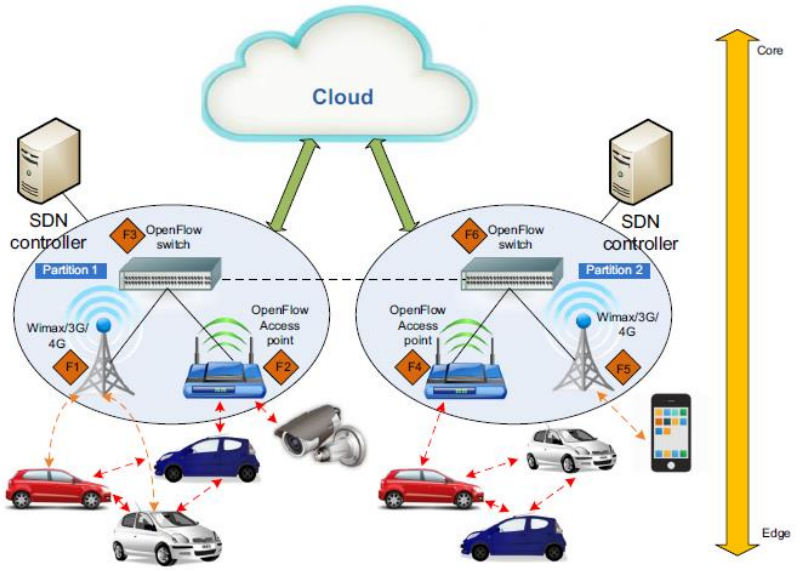

Figure 20 - SDN architecture for IoT based on Fog computing [70]

Chapter 3 – Proposed Solution

This chapter discusses the system architecture of the proposed solution, including some implementation details.

3.1. Architecture

As seen in the previous chapters, different technologies can be chosen in order to face the challenges enforced by IoT.

In this thesis it is proposed an architecture (Figure 21) in fog computing, using a system that combines SDN with virtualization, which is going to be used in an IoT scenario. As mentioned in the Literature Review, the SDN is organized into three planes: data plane, control plane and application plane. The data plane, composed by a switch, communicates with an SDN Controller through the Southbound API. In the application plane there is a Broker created to analyze messages and to request a container, that is started up when necessary. This Broker communicates with the controller through the Northbound API.

In the beginning, the container is not running in order to enhance the system sustainability and the energy efficiency. Whenever the system detects traffic to a certain container, it needs to guarantee that the container is started up, so it can receive and process messages.

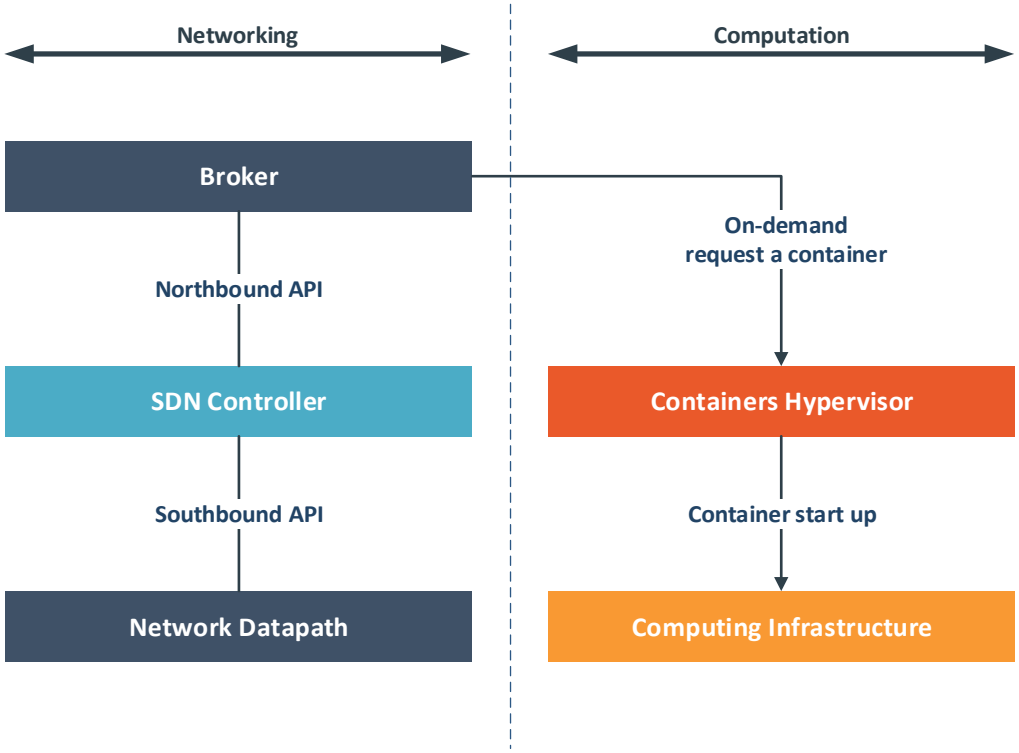

Figure 21 - Proposed Architecture

3.2. Deployment

As seen in chapter two, there are different SDN controllers, as well as containers and IoT communication protocols. This thesis is mainly focused on the Ryu controller. Many of its advantages include the support of numerous southbound protocols, comprising the latest version of OpenFlow, but mainly the possibility to write a Ryu application from scratch using Python. Three Ryu applications were written to allow the study of the different controller's behavior – reactive, hybrid and proactive mode. In each application, there is a websocket connection to a Broker, which was also written from the beginning and in Python to analyze messages and start the containers. Regarding containerization, Linux containers were chosen as they are a lightweight option. They allow the creation of a virtualized environment as closed as a standard Linux installation and authorize various applications to be installed and used in a single container. For the communication between containers, MQTT-SN is the most suitable application-layer protocol for this scenario, since it was created for being applied in sensor network, for using UDP and thus not requiring the establishment of a session, and for having a low message and header size. This protocol is yet not highly adopted, however it brings an opportunity since it has an offline keep-alive that allows devices to go to a sleeping state when they are not required, and then the messages are received afterwards, when they are activated.

Figure 22 represents the proposed architecture in detail. This architecture is composed by three linux containers, one Ryu Controller and a Websocket Broker. The LXC1 and LXC2 simulates IoT sensors and, as said in the previous section, in the beginning they are not running. The LXC3 is the container which has installed the MQTT-SN Broker. This container is always running as it simulates client that wants to request values from the sensors. Thus, the LXC3 sends an MQTT-SN message to one of the LXC1 or LXC2 sensors (step 1). When the OVS s1 receives the packet, it will check if there is a forwarding rule in the OpenFlow Table. If it finds a match entry, it will send the packet to the destination container (step 2). However, if there is no match in the flow table, a *packet_in* message is sent to the controller asking which action should be taken (step 2). Through a websocket connection, the Ryu controller sends the message to the Broker (step 3) so it can analyze and activate the correspondent container (step 4). Once it receives the feedback notifying that the container is running (step 5), the Broker informs the controller (step 6) so it can send a flow-entry to the OVS s1 containing the switch output port from where the packet should be forwarded (step 7). At least, the OVS s1 sends the packet to the correspondent sensor container.

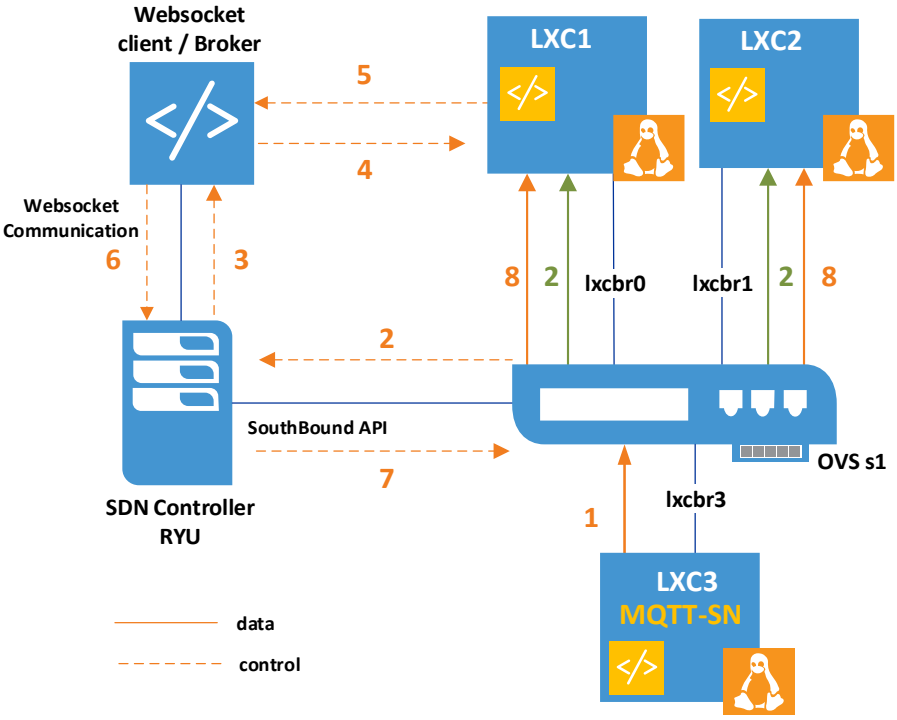

Figure 22 – Detailed proposed architecture

Open vSwitch

Figure 23 and Figure 24 represent the network topology created through a shell script wrote with network namespaces, bridge control and Open vSwitch commands. The network topology is composed by one Open vSwitch (OVS s1), 2 namespace hosts (h1, h2) and links / bridges. The hosts were created with the intent to test the Ryu applications, mainly the switch flow entries, through ping and iperf commands. Even tough containers were previously created, it was necessary to connect them to the Open vSwitch through linux bridges. At last, the OVS s1 was configured with a static datapath-id (1), OpenFlow protocol with version 1.3 and a connection to the Ryu controller in the port TCP/6633, which corresponds to the default port of OpenFlow.

```
Available hosts:
h2 (id: 1)
h1 (id: 0)
Available switches:
0ef7afbe-bb86-4b0e-a065-f581f1d5a200
    Bridge "s1"
        Controller "tcp:127.0.0.1:6633"
        Port "b0-veth3"
            Interface "b0-veth3"
        Port "s1"
            Interface "s1"
                type: internal
        Port "h1-veth"
            Interface "h1-veth"
        Port "h2-veth"
            Interface "h2-veth"
        Port "b0-veth1"
            Interface "b0-veth1"
        Port "b0-veth2"
            Interface "b0-veth2"
    ovs_version: "2.9.2"
bridge name     bridge id               STP enabled     interfaces
lxcbr0          8000.c22b4e51f1a5       no              b0-eth1
lxcbr1          8000.d68466102605       no              b0-eth2
lxcbr2          8000.00163e000000       no              b0-eth3
```

Figure 23 - Network Topology Configuration

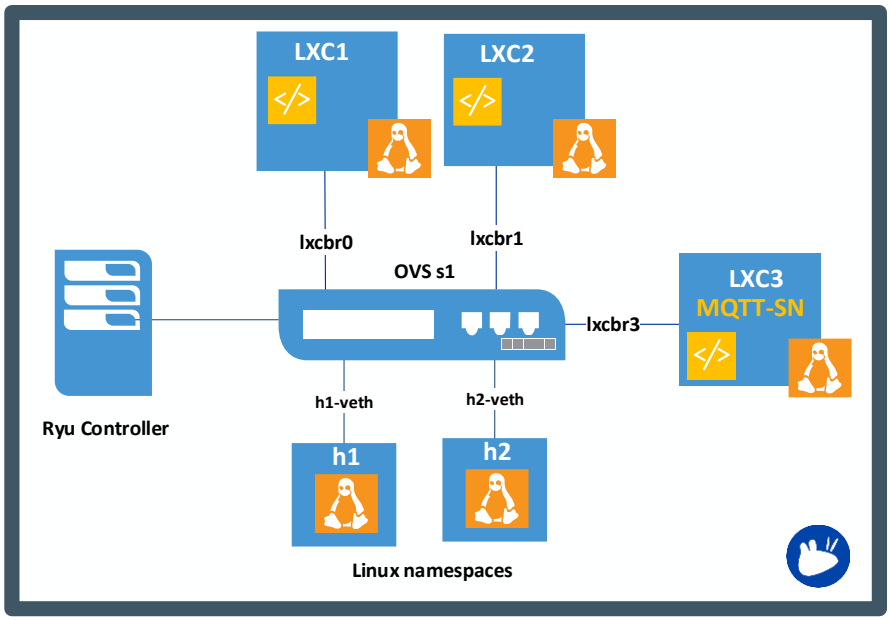

Figure 24 - Network Topology

Switch and Ryu's Controller Communication

As described above, three different Ryu applications were written. In the reactive mode, when the Ryu controller is started, a flow rule is installed into the switch to send a packet-in message to the controller whenever the switch receives a packet. It was used the class OFPActionOutput and the flag *OFPP_FLOOD* in the packet-out message, to specify the switch port that the packet should be forwarded, in this case, with this flag, the packet is forwarded to all ports except the one used to receive the packet.

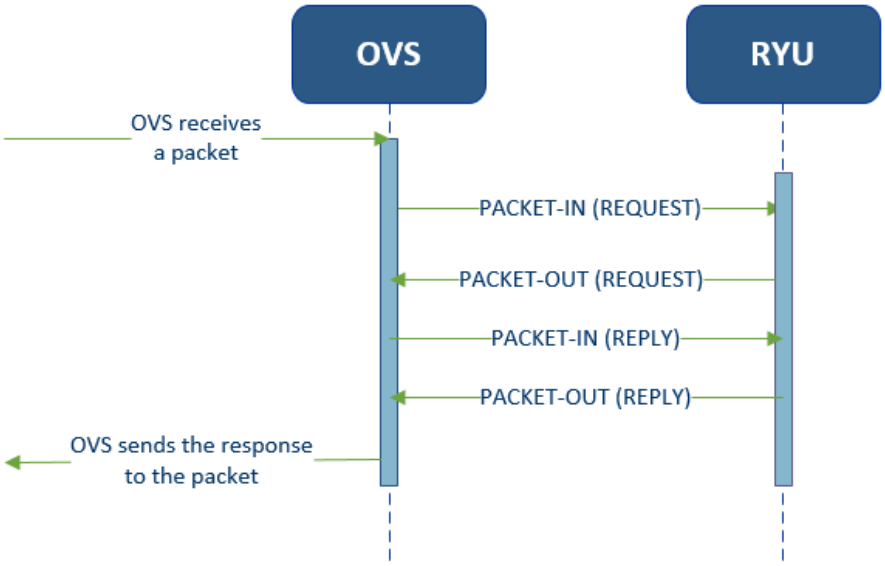

Figure 25 - Messages exchanged between Open vSwitch and Ryu Controller in the reactive mode

In the hybrid mode (Figure 26) the system behaves, in the beginning, in a similar way when compared to the reactive mode. The flow rule to send always a packet-in message to the controller is the only one installed into the switch when the Ryu controller is started. However, this only happens for the first packet of a specific flow. In this case, an *out_port* flag was used, so the Ryu controller could indicate through what port the packet should be forwarded. This new rule is installed into the switch flow table, avoiding future flooding and allowing a faster and more efficient transmission of the packets of the same flow by the switch.

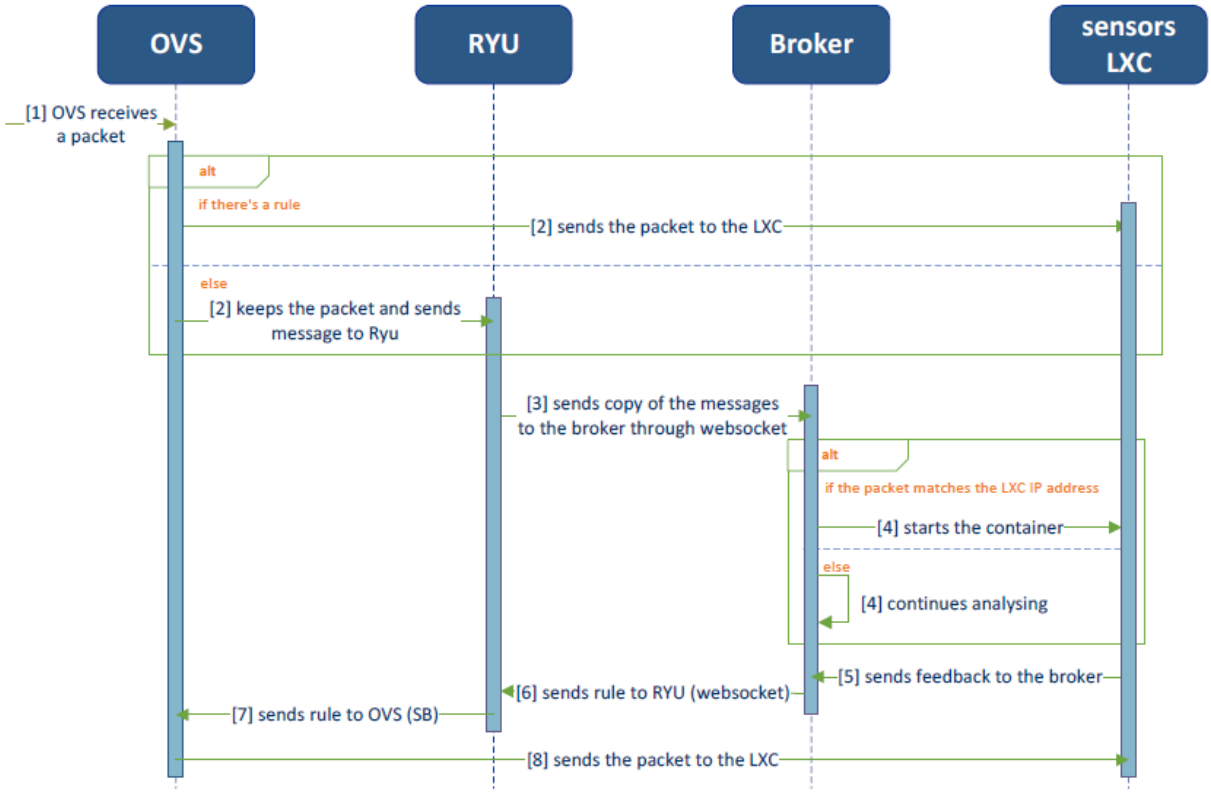

Figure 26 – Sequence diagram for hybrid mode

The proactive mode was written to install into the switch not only the “ask the controller” rule, but also the rules that indicate the port where the flow should be forwarded to reach pre-defined Linux containers. Using this application is expected to exist a lower number of packet-in / packet-out messages between the switch and the controller, allowing even better results when compared to the hybrid mode.

Broker

Every time the controller receives a packet, it sends a copy in a hexadecimal format to the Broker through a websocket connection. The Broker then analyses the message and extracts information about the source and destination IP and mac-address, ethertype, IP protocol and the port used. Figure 27 shows the hexadecimal messages and analysis made by the Broker.

```
{"params": ["{\"msg\": \"0xff 0xff 0xff 0xff 0xff 0xff 0x00 0x00 0x00 0x00 0x00 0x02 0x08 0x06 0x00 0x
01 0x08 0x00 0x06 0x04 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x02 0x0a 0x00 0x03 0x0c 0x00 0x00 0x00 0x00
 0x00 0x00 0x0a 0x00 0x03 0x0d\"}"], "jsonrpc": "2.0", "method": "send", "id": 165}
ARP Message
src IP: 10.0.3.12
dst IP: 10.0.3.13
LXC NOT FOUND!
```

Figure 27 – Hexadecimal message and analysis

Each Linux container has a static IP address as well as a MAC-address. When an ARP message is identified, and the destination IP address matches one of the LXC, the container is started. An information message appears if the container is already running or if the IP does not match any of the pre-defined containers.

MQTT-SN

In order to exchange MQTT-SN messages between the containers, it was necessary the installation of a Message Broker and a MQTT-SN Client. The Broker EMQ[1] was selected due to its versatility supporting several IoT protocols, such as MQTT, MQTT-SN and CoAP. It consists on an open source IoT MQTT message broker based on Erlang/OTP platform. To access the features of MQTT-SN, it is required the use of the EMQ-SN plugin[2]. EMQ recommends several MQTT-SN Clients, from which the MQTT-SN Tools[3] was chosen. These tools support some MQTT-SN features, *e.g.* as QoS -1, 0 and 1, publishing retained messages, short topic IDs, amongst others, having the disadvantage of not allowing the QoS 2.

A scenario, represented in Figure 28, was created where the LXC3 publishes messages under the topic *"*askTemp" and subscribes until it receives a message under the topic "sendTemp". When the LXC sensor receives a message requesting the temperature value, it then publishes in under the topic "sendTemp". Figure 28 shows the described scenario.

[1] https://docs.emqx.io/broker/v3/en/getstarted.html
[2] https://docs.emqx.io/broker/v2/en/mqtt-sn.html#emq-sn-plugin
[3] https://github.com/njh/mqtt-sn-tools

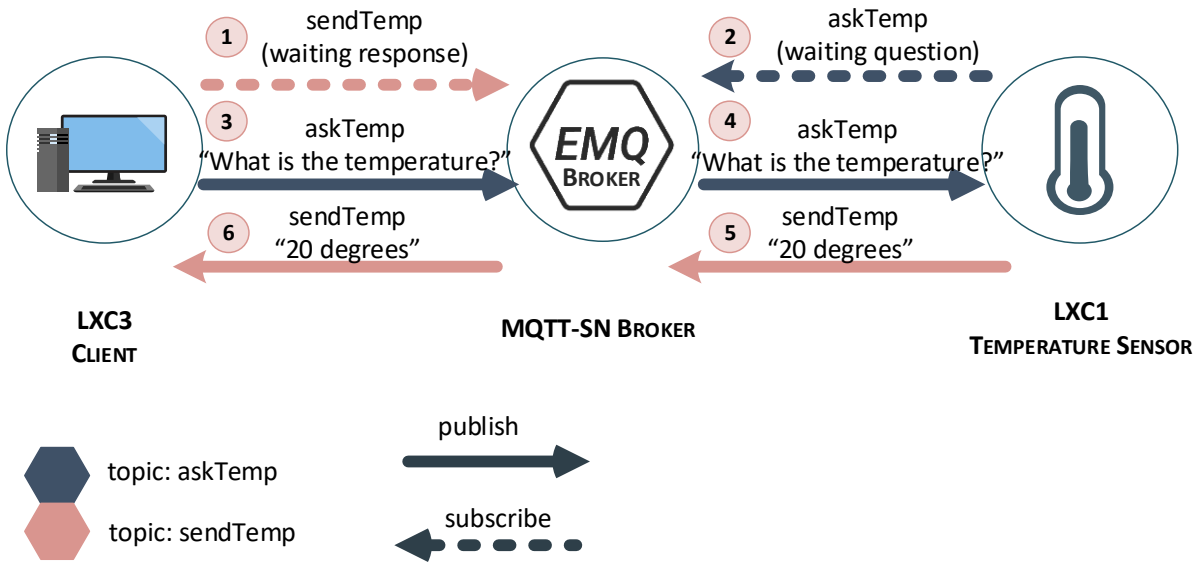

Figure 28 - MQTT-SN Scenario

Chapter 4 – Results and Discussion

This chapter describes all the performed tests that had aimed to evaluate the time required from activating the sensor containers to being able to communicate with them as well as the possibility of two containers to communicate with each other through MQTT-SN protocol. All the configuration, implementation and tests were developed in a Xubuntu VM hosted on VirtualBox.

The testing environment consists of a network topology, managed by a Ryu controller, and a Websocket Broker (represented in Figure 29). The network topology was built through a shell script with network namespaces, bridge control and Open vSwitch commands, leading to the creation of an Open vSwitch with OpenFlow v1.3 and two hosts with the respective connections. A logical TCP connection was also created between the OVS and the Ryu controller. Linux containers were previously created with Ubuntu OS through lxc commands and configured with a static IP and MAC-address and the indication on which Linux bridge they connect. LXC3, which is the container implemented with the MQTT-SN Broker, is the only container that is started up with the network topology script.

A Broker script was written in Python to decode hexadecimal messages and to extract essential information from the packet's header. When an ARP message is sent to the Broker and the destination IP matches one of the container's IP address, that container is started through the execution of lxc commands inside the script. The controller sends messages to the Broker through a websocket connection established between them.

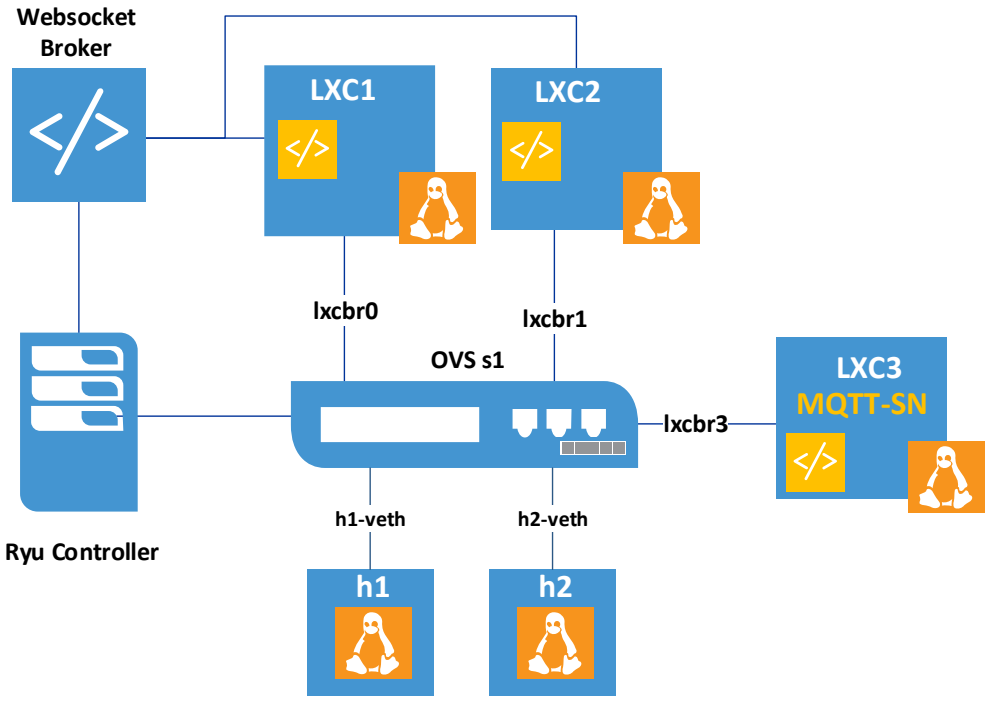

Figure 29 - Architecture topology

Performance and functional tests were executed and analyzed to evaluate the activation of the sensor containers and the communication between them through MQTT-SN protocol.

4.1. Performance Tests

As mentioned in the previous chapter, three Ryu controller's applications were written to assess the impact of the controller's behavior on the time required to process messages and start up the containers until they start communicating with each other. Each application is related to a controller's behavior.

The Ryu controller's applications were applied in two scenarios:

1. Communication between a network namespace and a container
2. Communication between containers

The results for these two scenarios were obtained by sending four ICMP packets. In the first scenario, the host h1 sends ICMP packets to the sensor's containers (LXC1 and LXC2), whereas in the second scenario this communication is started by the LXC3. A sample of 50 results was collected for each sensor container within each behavior mode.

Reactive Mode

In the reactive mode, a flow rule is installed into the switch with the indication to send a *packet-in* message to the controller whenever the switch receives a packet, *i.e.*, each time the host1 and LXC3 ping the sensor containers, a *packet-in* message is sent to the controller. In this case, no rule is installed into the switch flow table.

Observing Figure 30, it is possible to see that, in general, the first scenario had better response times than the second one. This is due to network namespaces being a Linux kernel feature that provides resource isolation, whereas Linux containers have yet a management interface, that interacts with the kernel components and provides tools for containers creation and management. It is still possible to highlight that the average response time of the first packet in both scenarios is higher when compared to the other three. This is due to the fact that in the first packet, not only the ICMP request and reply packets are considered, but also the ARP messages and the Linux container startup, which differs this packet from the others. As it is possible to see in Figure 31, the first ARP message (first *packet-in* and *packet-out*) lasted 506ms, whereas the second ARP message lasted 109ms. The overtime expressed by the first ARP message is associated with the activation of the Linux container, which in average launches in approximately 150 to 350ms. Only after the websocket Broker script starts the container, the ICMP packet can reach its destination.

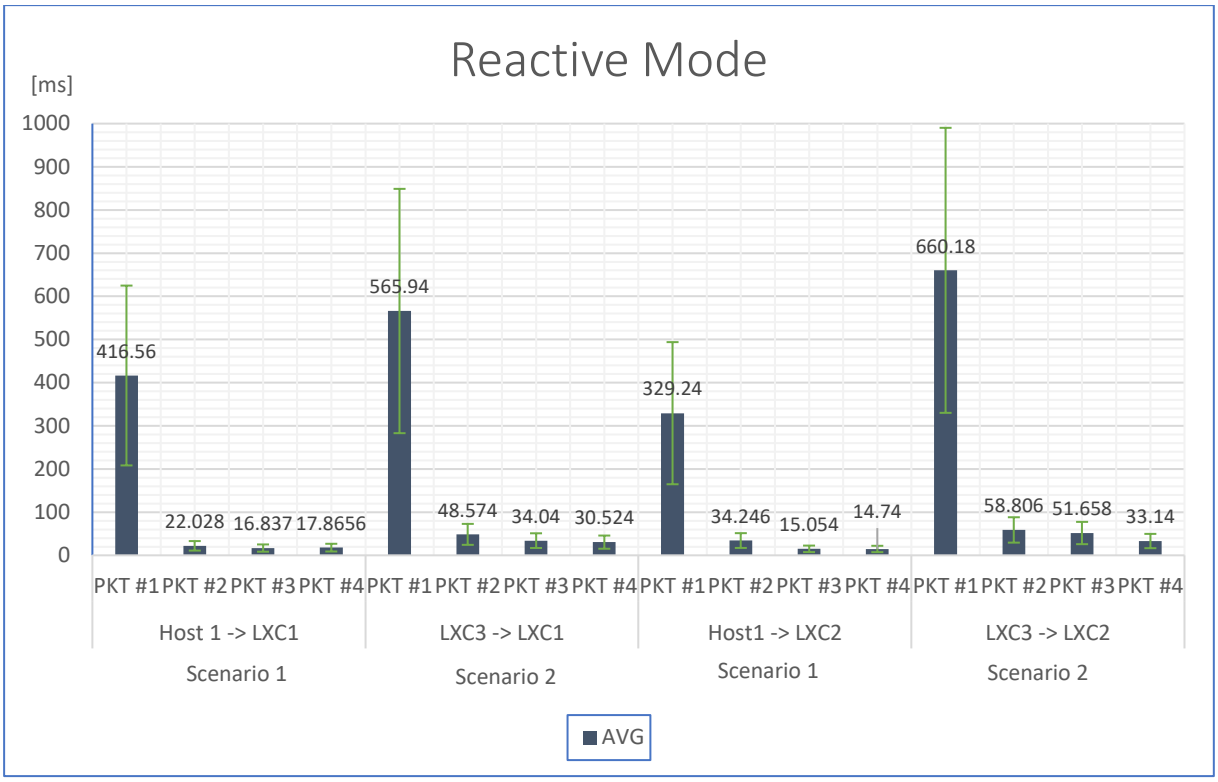

Figure 30 - Reactive Mode - Scenario 1 and 2

Time	Source	Destination	Protocol	Info
REF	127.0.0.1	127.0.0.1	OpenFlow	Type: OFPT_PACKET_IN
0.505992153	127.0.0.1	127.0.0.1	OpenFlow	Type: OFPT_PACKET_OUT
REF	127.0.0.1	127.0.0.1	OpenFlow	Type: OFPT_PACKET_IN
0.108769250	127.0.0.1	127.0.0.1	OpenFlow	Type: OFPT_PACKET_OUT

Figure 31 - Packet-In and Packet Out ARP Messages (Scenario 2)

As mentioned above, every time a packet arrives at the switch, a *packet-in* message is sent back to the controller, which in turn responds with a *packet-out* message indicating the port from where the packet should be sent. Thus, for every ICMP packet sent, a respective *packet-in* and *packet-out* is generated, as represented in Figure 31 and Figure 33, which is also the reason why packets 2 to 4 have high response time. The ICMP messages were exchanged during approximately 270ms.

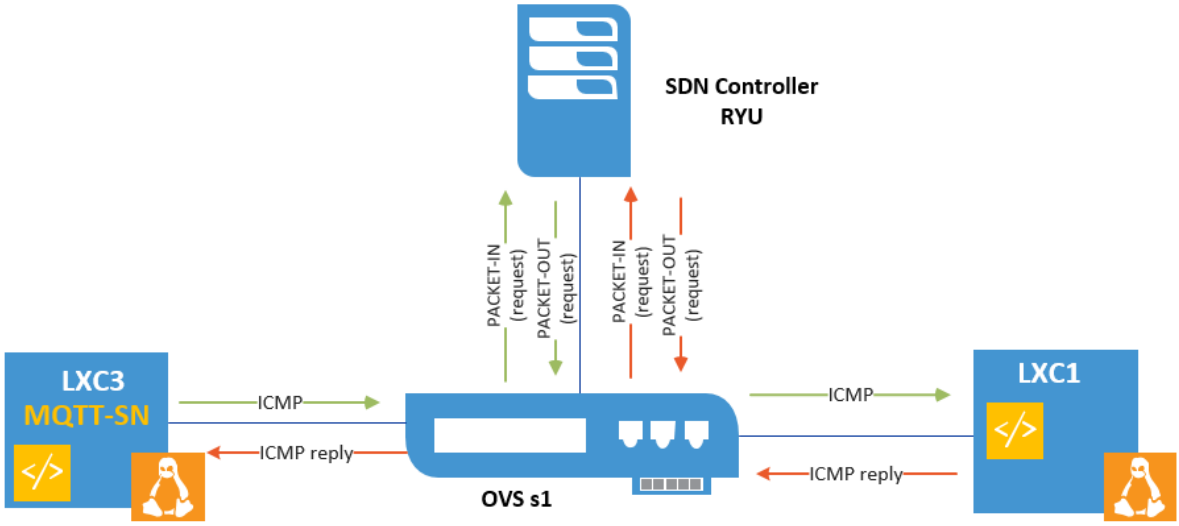

Figure 32 - ICMP traffic in the reactive mode (Scenario 2)

No.	Time	Source	Destination	Protocol	Length	Info
755	6.325028111	127.0.0.1	127.0.0.1	OpenFlow	206	Type: OFPT_PACKET_IN
762	6.343562632	127.0.0.1	127.0.0.1	OpenFlow	204	Type: OFPT_PACKET_OUT
763	6.344767027	127.0.0.1	127.0.0.1	OpenFlow	206	Type: OFPT_PACKET_IN
769	6.388728494	127.0.0.1	127.0.0.1	OpenFlow	204	Type: OFPT_PACKET_OUT
803	6.719252284	127.0.0.1	127.0.0.1	OpenFlow	206	Type: OFPT_PACKET_IN
809	6.738291918	127.0.0.1	127.0.0.1	OpenFlow	204	Type: OFPT_PACKET_OUT
811	6.738685229	127.0.0.1	127.0.0.1	OpenFlow	206	Type: OFPT_PACKET_IN
817	6.799100671	127.0.0.1	127.0.0.1	OpenFlow	204	Type: OFPT_PACKET_OUT
836	7.722469869	127.0.0.1	127.0.0.1	OpenFlow	206	Type: OFPT_PACKET_IN
849	7.777495615	127.0.0.1	127.0.0.1	OpenFlow	204	Type: OFPT_PACKET_OUT
853	7.779045074	127.0.0.1	127.0.0.1	OpenFlow	206	Type: OFPT_PACKET_IN
860	7.826565396	127.0.0.1	127.0.0.1	OpenFlow	204	Type: OFPT_PACKET_OUT
871	8.724667869	127.0.0.1	127.0.0.1	OpenFlow	206	Type: OFPT_PACKET_IN
879	8.734664586	127.0.0.1	127.0.0.1	OpenFlow	204	Type: OFPT_PACKET_OUT
881	8.734923260	127.0.0.1	127.0.0.1	OpenFlow	206	Type: OFPT_PACKET_IN
886	8.751231896	127.0.0.1	127.0.0.1	OpenFlow	204	Type: OFPT_PACKET_OUT

Figure 33 - Packet-In and Packet Out ICMP Messages (Scenario 2)

```
control is pinging sensorTemp:
PING 10.0.3.21 (10.0.3.21) 56(84) bytes of data.
64 bytes from 10.0.3.21: icmp_seq=1 ttl=64 time=674 ms
64 bytes from 10.0.3.21: icmp_seq=2 ttl=64 time=80.2 ms
64 bytes from 10.0.3.21: icmp_seq=3 ttl=64 time=106 ms
64 bytes from 10.0.3.21: icmp_seq=4 ttl=64 time=26.9 ms

--- 10.0.3.21 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 26.923/222.028/674.739/262.930 ms
*****
control is pinging sensorHum:
PING 10.0.3.22 (10.0.3.22) 56(84) bytes of data.
64 bytes from 10.0.3.22: icmp_seq=1 ttl=64 time=677 ms
64 bytes from 10.0.3.22: icmp_seq=2 ttl=64 time=121 ms
64 bytes from 10.0.3.22: icmp_seq=3 ttl=64 time=46.0 ms
64 bytes from 10.0.3.22: icmp_seq=4 ttl=64 time=59.4 ms

--- 10.0.3.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3012ms
rtt min/avg/max/mdev = 46.075/226.252/677.729/262.218 ms
```

Figure 34 – Reactive Mode - Ping from LXC3 (control) to LXC1 (sensorTemp) and LXC2 (sensorHum)

Hybrid Mode

As mentioned in the previous chapter, when the Ryu-hybrid mode application is executed, a rule is installed in the switch, with the indication to send a *packet-in* message to the controller if there is no match in the switch flow table. In Figure 35 it is possible to observe that the response time of the 1st ICMP packets is similar to the ones obtained in the reactive mode, as both have started with that only rule in the flow table.

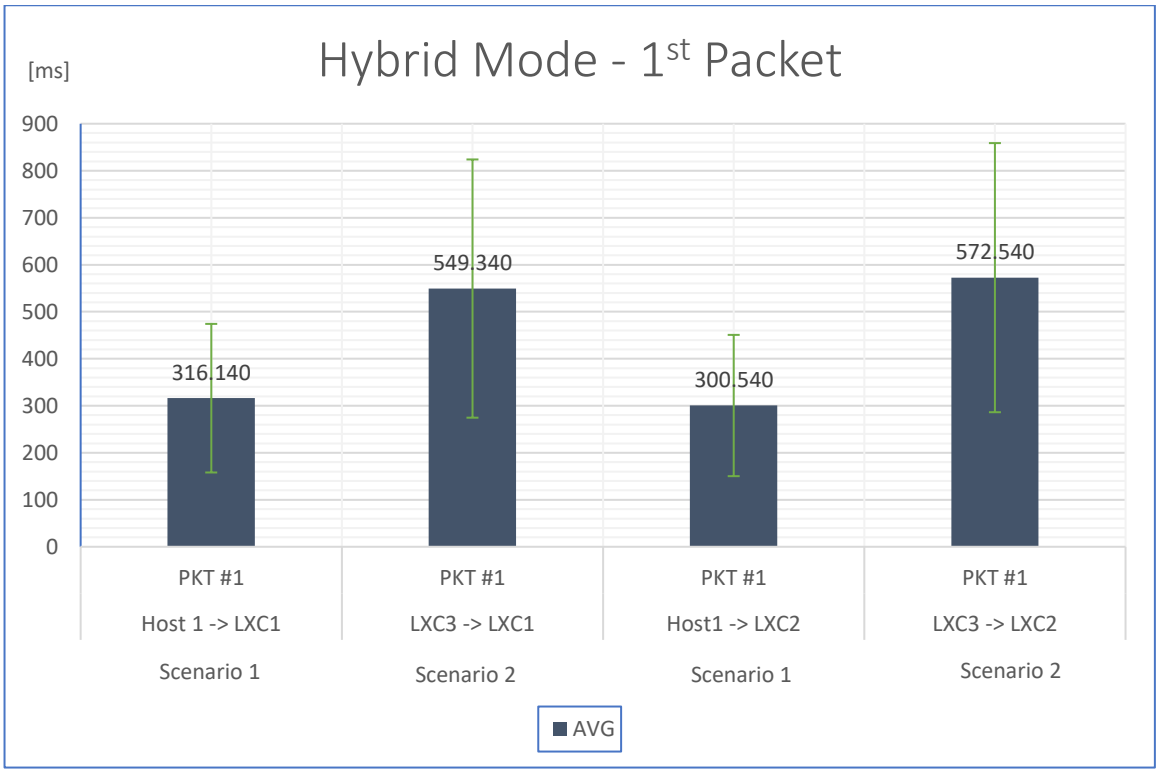

Figure 35 – Hybrid Mode - Scenario 1 and 2, 1st packet

However, looking at Figure 36, the next ICMP packets response time are very low when compared to the reactive mode. This is due to the installation of a rule by the controller into the switch with the indication of the port from where the traffic should be forwarded (represented in Figure 37). Thus, from the second packet forward, the switch doesn't have to send a message to the controller as it already knows the port, sending the packet straight to the port where the destination container is connected.

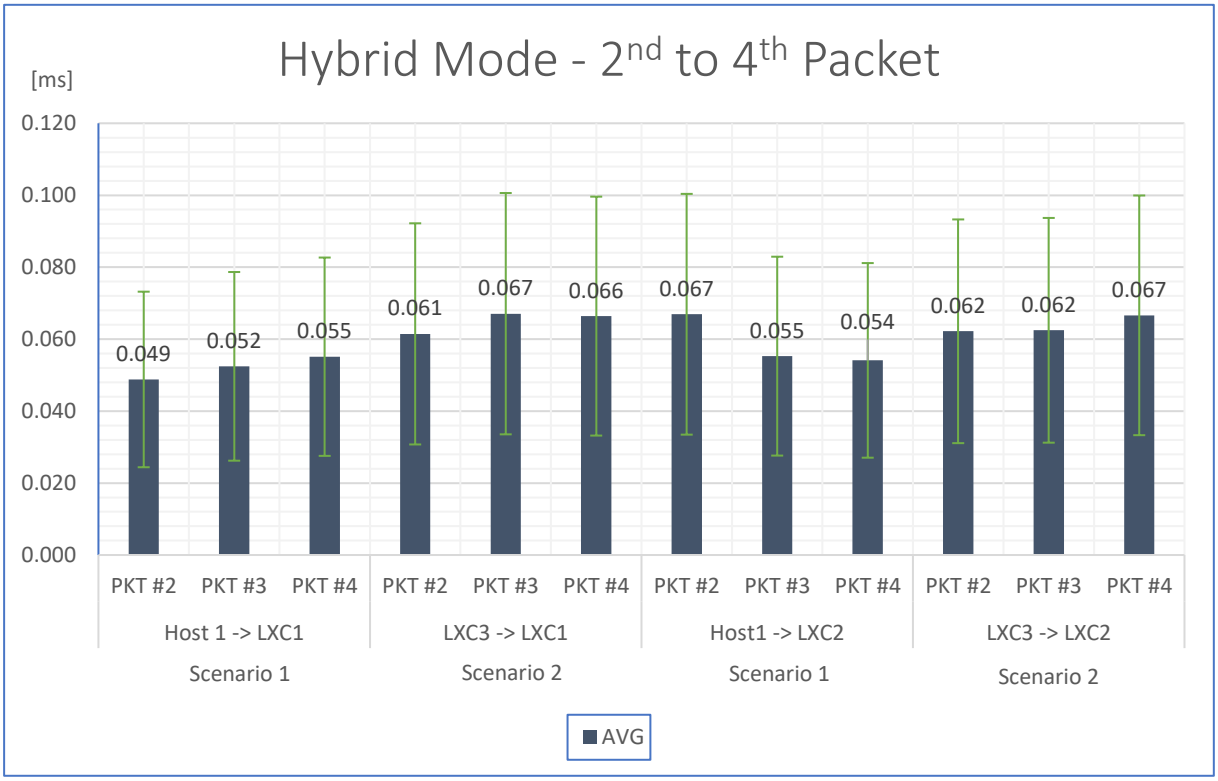

Figure 36 - Hybrid Mode - Scenario 1 and 2, 2nd to 4th packet

```
control is pinging sensorHum:
PING 10.0.3.22 (10.0.3.22) 56(84) bytes of data.
64 bytes from 10.0.3.22: icmp_seq=1 ttl=64 time=654 ms
64 bytes from 10.0.3.22: icmp_seq=2 ttl=64 time=0.089 ms
64 bytes from 10.0.3.22: icmp_seq=3 ttl=64 time=0.068 ms
64 bytes from 10.0.3.22: icmp_seq=4 ttl=64 time=0.119 ms

--- 10.0.3.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3035ms
rtt min/avg/max/mdev = 0.068/163.584/654.060/283.176 ms
*****
Switch s1 has the following flow rules:
 cookie=0x0, duration=12.415s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_dst=01:80:c2:
00:00:0e,dl type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=7.653s, table=0, n_packets=5, n_bytes=434, priority=1,in_port="b0-veth1",
dl_src=00:00:00:00:00:21,dl_dst=00:00:00:00:00:26 actions=output:"b0-veth3"
 cookie=0x0, duration=7.612s, table=0, n_packets=4, n_bytes=336, priority=1,in_port="b0-veth3",
dl_src=00:00:00:00:00:26,dl_dst=00:00:00:00:00:21 actions=output:"b0-veth1"
 cookie=0x0, duration=3.440s, table=0, n_packets=4, n_bytes=392, priority=1,in_port="b0-veth2",
dl_src=00:00:00:00:00:22,dl_dst=00:00:00:00:00:26 actions=output:"b0-veth3"
 cookie=0x0, duration=3.400s, table=0, n_packets=3, n_bytes=294, priority=1,in_port="b0-veth3",
dl_src=00:00:00:00:00:26,dl_dst=00:00:00:00:00:22 actions=output:"b0-veth2"
 cookie=0x0, duration=12.514s, table=0, n_packets=114, n_bytes=15942, priority=0 actions=CONTRO
LLER:65535
*****
```

Figure 37 - Hybrid Mode - Ping from LXC3 to LXC2 and switch flow table rules

Proactive Mode

To the proactive mode application, a function was added in order to add rules into the switch flow table. Thus, when the Open vSwitch is created and is connected to the Ryu controller, those rules are pre-installed in it, indicating the port to where the traffic should be forwarded. This added function was required to avoid unnecessary messages going to the controller, however, ARP messages rules could not be installed into the switch as they are needed to activate the containers. When the Broker script identifies an ARP message, it analyses the destination container and starts it up. The installation of these type of rules prevents the message to reach the Broker and therefore, makes it impossible to start the containers.

In Figure 38 it is possible to notice a huge improvement from the scenario 1 since the average response time of the 1st packet decreased from approximately 300ms to 200ms. Good response time results were expected in this proactive mode when compared to the hybrid mode, as the installation of forwarding rules reduces the number of messages exchanged between the switch and the controller.

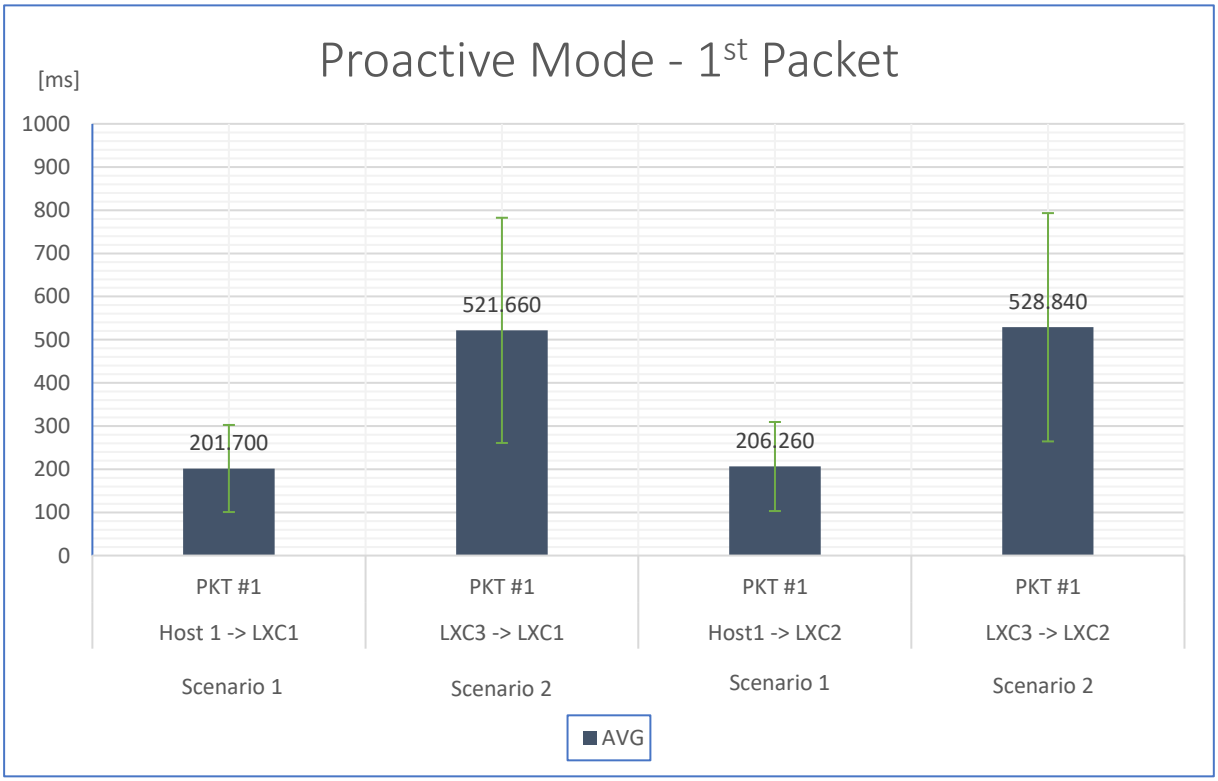

Figure 38 - Proactive Mode - Scenario 1 and 2, 1st packet

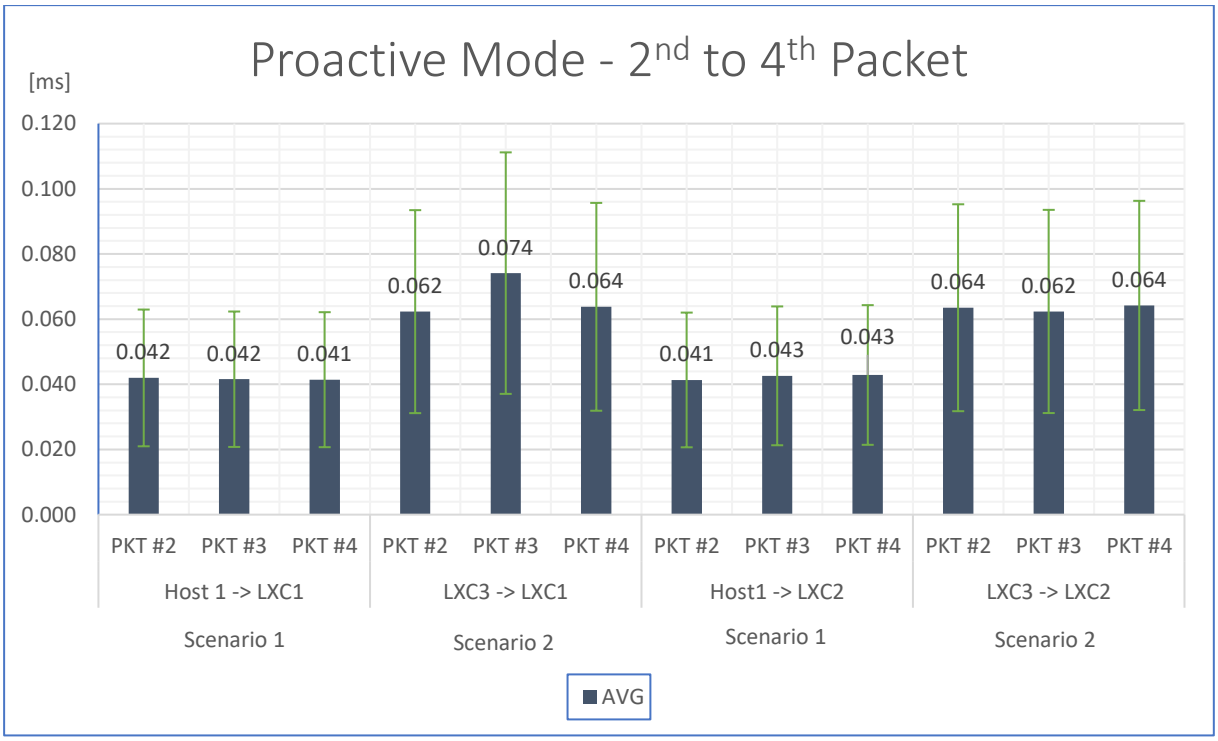

Figure 39 - Proactive Mode - Scenario 1 and 2, 2nd to 4th packet

Results Comparison

Table 7 details the comparison between the three controller's behavior: reactive, hybrid and proactive.

Table 7 - Performance Tests Comparison

Controller's Behavior	PKT	Scenario 1		Scenario 2	
		Host 1 -> LXC1	Host 1 -> LXC2	LXC3 -> LXC1	LXC3->LXC2
Reactive	1	**416.560**	**329.240**	**565.940**	**660.180**
	2	22.028	34.246	48.574	58.806
	3	16.837	15.054	34.040	51.658
	4	17.866	14.740	30.524	33.140
Hybrid	1	**316.140**	**300.540**	**549.340**	**572.540**
	2	0.049	0.067	0.061	0.062
	3	0.052	0.055	0.067	0.062
	4	0.055	0.054	0.066	0.067
Proactive	1	**201.700**	**206.260**	**521.660**	**528.840**
	2	0.042	0.041	0.062	0.064
	3	0.042	0.043	0.074	0.062
	4	0.041	0.043	0.064	0.064

Observing Table 7, it is possible to notice that the proactive mode obtained the best time response results. Looking at the first packet, there is a time decrease from the hybrid mode to the proactive mode, especially in scenario 1. This is due to the pre-installation rules in the Open vSwitch that reduces the number of exchanged messages between the switch and the controller. Regarding the second, third and fourth packets, it is possible to observe that both the proactive and hybrid modes obtained similar results, as the packets were forwarded directly to the sensor container. Overall, it is possible to see that the worst time response results were obtained by the reactive mode in all four packets. Having only the rule to send packets to the controller, leaded to an increase of flow messages, as all the packets had to be analyzed by the controller so they could be forwarded to the intended container.

4.2. Functional Tests

The functional tests consisted in the communication between containers using the MQTT-SN protocol.

Two scripts were written with the purpose to run inside the LXC3 and sensor containers. The LXC3 simulates the client that requests values from the sensors. Thus, the *askTemperature.py* simulates the question from the client “What is the temperature?” and publishes this question into the topic with the name *askTemp*.

The containers then wait for the answer. In the sensor’s containers, *e.g.* LXC1, the script simulates the IoT sensor sending the value requested by the LXC3. Hence, the *sendTemperature.py* was written to subscribe the topic *askTemp* and to publish the temperature value in the *sendTemp*. This topic is already being subscribed by the LXC3 and when the LXC1 sends the value, the LXC3 immediately received it. Figure 40 represents publishing a message by the LXC3 and then the followed response, whereas in Figure 41, the LXC1 first subscribes the topic and then the LXC3 publishes the message.

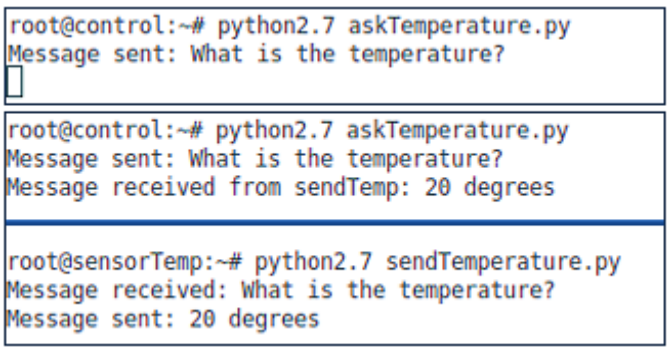

Figure 40 - Message exchanged through MQTT-SN (LXC3 first publishes message)

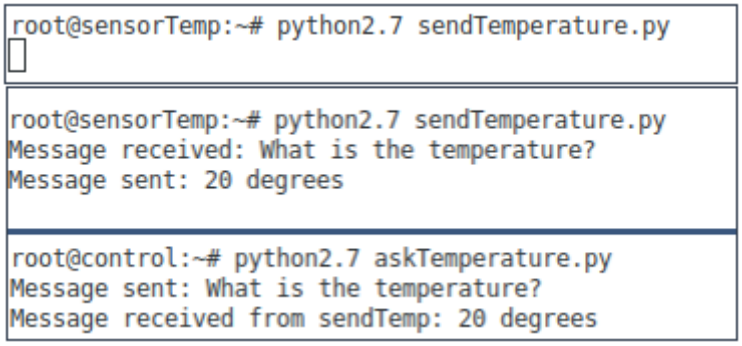

Figure 41 - Message exchanged through MQTT-SN (LXC1 first subscribes topic)

The main intention of this test was to activate containers by sending an MQTT-SN message. However due to implementation problems and time issues, this was not accomplished and therefore in this case, the LXC3 first sends an ICMP to activate the container and then the scripts are executed.

Chapter 5 - Conclusions and Future Work

5.1. Conclusions

The main goal of this thesis was the design and evaluation of an architecture that used a software-defined networking approach to efficiently activate fog computational resources on demand, whenever required by IoT or sensor networks applications.

The exponential increase of IoT devices creates new challenges such as scalability, mobility, heterogeneity and security. Traditional networks have become inefficient to deal with these issues as well as the huge volume of data. Thus, software-defined networking has emerged as being flexible and scalable, providing a centralized logical control of the network devices and solving complexed technologies such as fog computing.

To address these challenges, this thesis suggested the use of several emerging technologies including software-defined networking, containerization, fog computing, and sensor virtualization. For this purpose, a software-defined system was implemented, which activates the containers when required, managed by a Ryu SDN Controller and an intelligent Broker. This Broker decodes hexadecimal messages sent by the Controller through a websocket connection and extracts information from the packet's header. When it receives an ARP message and the destination IP matches one of the container's IP address, the Broker starts that container.

Performance and functional tests were performed to evaluate the time required from activating the sensor containers to being able to communicate with them as well as the time required for the communication between the containers through the MQTT-SN protocol.

The performance test was executed with three different SDN controller code versions in order to know how long it takes for the container to start communicating. This was achieved by evaluating the time response of four ICMP packets sent by Host 1 and LXC3 to the sensor containers. The best time response results were obtained by the proactive mode; for instance, when analyzing the first packet, it was possible to conclude that the response time decreased from the hybrid mode to the proactive mode, due to pre-installation rules in the Open vSwitch. Regarding the second to the fourth packets, similar results were obtained for both the proactive and hybrid modes. Overall, it was possible to conclude that the worse time response results were obtained by the reactive mode in all four packets. This was caused by the fact that the Open vSwitch only had the rule to send packets to the Controller, leading to an increase of flow messages. All the obtained results were within an acceptable range concerning an IoT environment, considering the relatively low delay on the response times observed under these tests.

The functional test consisted in the communication between containers through the MQTT-SN protocol. This protocol was chosen over the MQTT as it uses UDP, which is essential to the communication with offline IoT computational resources. It has been demonstrated that it is possible for the LXC3 to publish a message while the sensor's containers are offline, and the message subscription when the containers are activated. However, one of the main goals in terms of its implementation was based on the

application of both tests simultaneously, *i.e.* the activation of containers though an MQTT-SN message and the immediate messages exchanged. This goal was not possible to accomplish during this thesis due to time related issues for its implementation.

The proof-of-concept study entailed the following question, which served as bases for this thesis:

> "Is it possible to efficiently perform on-demand activation of IoT computing resources using a software-defined system?"

Overall, this thesis helped contributing to fill the gaps in the area of IoT or sensor networks, concerning the design and implementation of an architecture that performed on-demand activation of offline IoT fog computing resources by using an SDN controller, containerization and sensor virtualization.

5.2. Future Work

This thesis opens the way for further research featuring software-defined networking solutions for managing fog computing resources in sensor networks. Proposed future research includes the study and possible change of the current designed architecture or scripts in order to activate the containers by publishing an MQTT-SN message. Another suggestion is the exploration of MQTT-SN quality-of-service, which was not approached in this thesis. The MQTT-SN client was used with the QoS default value of 0, *i.e.* at most once. The utilization of real sensors could be implemented in this architecture, in order to obtain more accurate results and to study values of energy consumption.

Chapter 6 – References

[1] R. Rangel, “Inovação em Pauta: Certificação digital, um caminho sem volta,” pp. 4–7, 2014.

[2] C. MacGillivray, V. Turner, and D. Lund, “Worldwide Internet of Things (IoT) 2013–2020 Forecast: Billions of Things, Trillions of Dollars,” 2013.

[3] P. Hu, S. Dhelim, H. Ning, and T. Qiu, “Survey on fog computing: architecture, key technologies, applications and open issues,” *Journal of Network and Computer Applications*, vol. 98, pp. 27–42, 2017.

[4] A. C. Baktir, A. Ozgovde, and C. Ersoy, “How Can Edge Computing Benefit from Software-Defined Networking: A Survey, Use Cases , and Future Directions,” *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2359–2391, 2017.

[5] S. Sezer, S. Scott-Hayward, P. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, “Are we ready for SDN? Implementation challenges for software-defined networks,” *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[6] C. Aggarwal and K. Srivastava, “Securing IOT devices using SDN and edge computing,” in *Proceedings on 2016 2nd International Conference on Next Generation Computing Technologies, NGCT 2016*, 2017, pp. 877–882.

[7] D. Kreutz, F. M. V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-Defined Networking: A Comprehensive Survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[8] K. Ogawa, K. Kanai, K. Nakamura, H. Kanemitsu, J. Katto, and H. Nakazato, “IoT Device Virtualization for Efficient Resource Utilization in Smart City IoT Platform,” in *2019 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2019*, 2019.

[9] R. Buyya and A. V. Dastjerdi, *Internet of Things: Principles and Paradigms*. 2016.

[10] K. K. Patel and S. M. Patel, “Internet of Things-IOT: definition, characteristics, architecture, enabling technologies, application & future challenges,” *International Journal of Engineering Science and Computing*, vol. 6, no. 5, pp. 6122–6131, 2016.

[11] T. Guarda, M. Leon, M. F. Augusto, L. Haz, M. De La Cruz, W. Orozco, and J. Alvarez, “Internet of Things challenges,” in *Iberian Conference on Information Systems and Technologies, CISTI*, 2017.

[12] H. Aldowah, S. Ul Rehman, and I. Umar, “Security in internet of things: Issues, challenges and solutions,” in *Advances in Intelligent Systems and Computing*, 2019, pp. 396–405.

[13] N. N. Srinidhi, S. M. Dilip Kumar, and K. R. Venugopal, “Network optimizations in the Internet of

Things: A review," *Engineering Science and Technology, an International Journal*, vol. 22, no. 1, pp. 1–21, 2019.

[14] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate, "A Survey on Application Layer Protocols for the Internet of Things," *Transaction on IoT and Cloud Computing*, vol. 3, no. 1, pp. 9–18, 2015.

[15] Kaa Project, "What is the Internet of Things Platform," 2015. [Online]. Available: https://www.kaaproject.org/what-is-iot/. [Accessed: 18-Aug-2019].

[16] K. L. Lueth and J. Kotzorek, "IoT Platforms - The central backbone for the Internet of Things," 2015.

[17] S. Hamdani and H. Sbeyti, "A Comparative study of COAP and MQTT communication protocols," in *7th International Symposium on Digital Forensics and Security (ISDFS)*, 2019.

[18] V. Tirupathi and K. Sagar, "A Research on Interoperability Issues in Internet of Things at Application Layer," *International Journal of Recent Technology and Engineering*, vol. 8, no. 1S4, pp. 785–788, 2019.

[19] E. F. Silva, B. Jose Dembogurski, A. B. Vieira, and F. Henrique Cerdeira Ferreira, "IEEE P21451-1-7: Providing More Efficient Network Services over MQTT-SN," in *SAS 2019 - 2019 IEEE Sensors Applications Symposium, Conference Proceedings*, 2019.

[20] A. Stanford-Clark and H. L. Truong, "MQTT for sensor networks (MQTT-SN) protocol specification," *IBM*. 2013.

[21] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *2017 IEEE International Symposium on Systems Engineering, ISSE 2017 - Proceedings*, 2017.

[22] C. Moratelli, S. Johann, F. Hessel, and M. Neves, "Embedded virtualization for the design of secure IoT applications," in *Proceedings of the 2016 27th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype, RSP 2016*, 2016.

[23] Q. Duan and M. Toy, *Virtualized Software-Defined Networks and Services*. Norwood: Artech House, 2017.

[24] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, 2015.

[25] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support PaaS," in *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, 2014, pp. 610–614.

[26] Intel, "Container and Kernel-Based Virtual Machine (KVM) Virtualization for Network Function Virtualization (NFV)," 2015.

[27] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud*

Computing, vol. 1, no. 3, pp. 81–84, 2014.

[28] Linux, “Linux Containers - LXC.” [Online]. Available: https://linuxcontainers.org/lxc/introduction/. [Accessed: 21-Jul-2019].

[29] Á. Kovács, “Comparison of different linux containers,” in *2017 40th International Conference on Telecommunications and Signal Processing, TSP 2017*, 2017, pp. 47–51.

[30] Docker, *Docker for the Virtualization Admin*. 2016.

[31] Docker, “Docker overview.” [Online]. Available: https://docs.docker.com/engine/docker-overview/#what-can-i-use-docker-for. [Accessed: 24-Jul-2019].

[32] “State of the IoT 2018: Number of IoT devices now at 7B,” *IoT Analytics*, 2018. [Online]. Available: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/. [Accessed: 03-Jun-2019].

[33] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, “Network function virtualization: State-of-the-art and research challenges,” *IEEE Communications Surveys and Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[34] T. Kovacshazy, G. Wacha, T. Daboczi, C. Erdos, and A. Szarvas, “System architecture for Internet of Things with the extensive use of embedded virtualization,” in *4th IEEE International Conference on Cognitive Infocommunications, CogInfoCom 2013 - Proceedings*, 2013, pp. 549–554.

[35] S. Bose, A. Gupta, S. Adhikary, and N. Mukherjee, “Towards a Sensor-Cloud Infrastructure with Sensor Virtualization,” in *MSCC ’15 Proceedings of the Second Workshop on Mobile Sensing, Computing and Communication*, 2015, pp. 25–30.

[36] J. Ko, B.-B. Lee, K. Lee, S. G. Hong, N. Kim, and J. Paek, “Sensor Virtualization Module: Virtualizing IoT Devices on Mobile Smartphones for Effective Sensor Data Management,” *International Journal of Distributed Sensor Networks*, vol. 2015, pp. 1–10, 2015.

[37] X. Chen, L. Jiao, W. Li, and X. Fu, “Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing,” *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2016.

[38] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, and O. Rana, “Fog computing for the Internet of Things: A survey,” *ACM Transactions on Internet Technology*, vol. 19, no. 2, 2019.

[39] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, “All one needs to know about fog computing and related edge computing paradigms: A complete survey,” *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[40] OpenFog Consortium Architecture Working Group, “OpenFog Reference Architecture for Fog Computing,” 2017.

[41] P. Martinez-Julia and A. F. Skarmeta, “Empowering the Internet of Things with Software Defined Networking,” Geneva, Switzerland, 2014.

[42] L. Mamushiane, A. Lysko, and S. Dlamini, “A comparative evaluation of the performance of popular SDN controllers,” in *IFIP Wireless Days*, 2018, pp. 54–59.

[43] “Open Networking Foundation.” [Online]. Available: https://www.opennetworking.org/. [Accessed: 09-May-2019].

[44] H. A. Eissa, K. A. Bozed, and H. Younis, “Software Defined Networking,” in *19th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering, STA 2019*, 2019, pp. 620–625.

[45] S. Azodolmolky, *Software Defined Networking with OpenFlow*. Birmingham, Mumbai: Packt Publishing Ltd., 2013.

[46] Open Networking Foundation, “OpenFlow Switch Specification (Version 1.5.1).” 2015.

[47] Linux Foundation, “Open vSwitch.” [Online]. Available: http://www.openvswitch.org/. [Accessed: 15-May-2019].

[48] P. Krongbaramee and Y. Somchit, “Implementation of SDN Stateful Firewall on Data Plane using Open vSwitch,” in *Proceeding of 2018 15th International Joint Conference on Computer Science and Software Engineering, JCSSE 2018*, 2018, pp. 1–5.

[49] M. P. Fernandez, “Comparing OpenFlow controller paradigms scalability: Reactive and proactive,” in *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, 2013, pp. 1009–1016.

[50] The Linux Foundation, “OpenDaylight.” [Online]. Available: https://www.opendaylight.org/. [Accessed: 22-Apr-2019].

[51] Z. K. Khattak, M. Awais, and A. Iqbal, “Performance evaluation of OpenDaylight SDN controller,” in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, 2014, pp. 671–676.

[52] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, “SDN controllers: A comparative study,” in *Proceedings of the 18th Mediterranean Electrotechnical Conference: Intelligent and Efficient Technologies and Services for the Citizen, MELECON 2016*, 2016, pp. 1–6.

[53] S. Asadollahi, B. Goswami, and A. Gonsai, “Software Defined Network, Controller Comparison,” *Ijircce*, vol. 5, no. 2, pp. 211–217, 2017.

[54] C. Fernandez and J. L.Muñoz, *Software Defined Networking (SDN) with OpenFlow 1.3, Open vSwitch and Ryu*. 2016.

[55] Ryu SDN Framework Community, “Ryu Documentation.” [Online]. Available: http://osrg.github.io/ryu/resources.html#documentation. [Accessed: 02-Jun-2019].

[56] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, “Advanced study of SDN/OpenFlow controllers,” in *CEE-SECR ’13 Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, 2013, pp. 1–6.

[57] D. Erickson, “The Beacon OpenFlow controller,” in *HotSDN 2013 - Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013, pp. 13–18.

[58] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, “Feature-based comparison and selection of Software Defined Networking (SDN) controllers,” in *2014 World Congress on Computer Applications and Information Systems, WCCAIS 2014*, 2014, pp. 1–7.

[59] M. B. Al-Somaidai, “Survey of Software Components to Emulate OpenFlow Protocol as an SDN Implementation,” *American Journal of Software Engineering and Applications*, vol. 3, no. 6, pp. 74–82, 2014.

[60] S. Rowshanrad, V. Abdi, and M. Keshtgari, “Performance evaluation of sdn controllers: Floodlight and OpenDaylight,” *IIUM Engineering Journal*, vol. 17, no. 2, pp. 47–57, 2016.

[61] P. V. Tijare and D. Vasudevan, “The Northbound APIs of Software Defined Networks,” *International Journal of Engineering Sciences & Research Technology*, vol. 5, no. 10, pp. 501–513, 2016.

[62] S. Demirci and S. Sagiroglu, “Optimal placement of virtual network functions in software defined networks: A survey,” *Journal of Network and Computer Applications 147*, vol. 147, 2019.

[63] Y. Li and M. Chen, “Software-defined network function virtualization: A survey,” *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[64] M. Alenezi, K. Almustafa, and K. A. Meerja, “Cloud based SDN and NFV architectures for IoT infrastructure,” *Egyptian Informatics Journal*, vol. 20, no. 1, pp. 1–10, 2019.

[65] M. Ojo, D. Adami, and S. Giordano, “A SDN-IoT architecture with NFV implementation,” in *2016 IEEE Globecom Workshops, GC Wkshps 2016 - Proceedings*, 2016, pp. 1–6.

[66] L. Valdivieso, P. Ludeña-González, R. Torres, and L. Barona, “SDN/NFV Architecture for IoT Networks,” in *Proceedings of the 14th International Conference on Web Information Systems and Technologies*, 2018, pp. 425–429.

[67] I. Bedhief, M. Kassar, and T. Aguili, “SDN-based architecture challenging the IoT heterogeneity,” in *2016 3rd Smart Cloud Networks and Systems, SCNS 2016*, 2016, pp. 1–3.

[68] R. Morabito and N. Beijar, “Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies,” in *2016 IEEE International Conference on Sensing, Communication and Networking, SECON Workshops 2016*, 2016, pp. 1–6.

[69] Y. Xu, V. Mahendran, and S. Radhakrishnan, “SDN docker: Enabling application auto-docking/undocking in edge switch,” in *Proceedings - IEEE INFOCOM*, 2016, pp. 864–869.

[70] S. Tomovic, K. Yoshigoe, I. Maljevic, and I. Radusinovic, “Software-Defined Fog Network Architecture for IoT,” *Wireless Personal Communications*, vol. 92, no. 1, pp. 181–196, 2017.