Agnostic APIs for Operating Systems in Heterogeneous Internet of Things Endpoint Devices

Afonso Rodrigues Ferreira Oliveira Carneiro

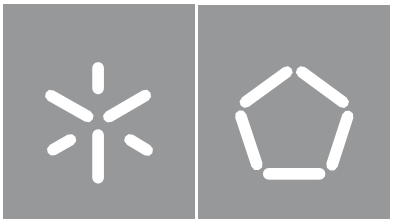
UMinho | 2020

Universidade do Minho
Escola de Engenharia

Afonso Rodrigues Ferreira Oliveira Carneiro

Agnostic APIs for Operating Systems in Heterogeneous Internet of Things Endpoint Devices

julho de 2020



Universidade do Minho
Escola de Engenharia

Afonso Rodrigues Ferreira Oliveira Carneiro

Agnostic APIs for Operating Systems in Heterogeneous Internet of Things Endpoint Devices

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Tiago Gomes

julho de 2020

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Agradecimentos

Este capítulo tem como finalidade agradecer a todas as pessoas que estiveram presentes durante a criação e desenvolvimento desta dissertação de mestrado.

Os meus sinceros agradecimentos aos meus companheiros de laboratório pelas conversas e gargalhadas. Obrigado por tornarem esta jornada ainda mais especial.

Quero aproveitar para poder agradecer às pessoas mais próximas de mim. Mãe, obrigado por seres quem és. Obrigado pelo apoio incondicional que só espero um dia poder dar aos meus filhos. És sem dúvida insubstituível. Miguel, irmão, obrigado por me ensinares enquanto tu também aprendes. Sem dúvida que a jornada da vida é professora, e acredito plenamente que tu serás o melhor aluno. Alexandre, obrigado por seres um irmão mais velho em todos os sentidos. Para o João Reis, um obrigado especial pela amizade, pelo apoio e pelos conselhos dados ao longo destes 6 anos.

Para o meu avô e para a minha avó, obrigado por tudo.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Conectar uma grande quantidade de dispositivos *low-end* à *Internet of Things (IoT)* apresenta desafios que devem ser considerados desde os estágios iniciais de desenvolvimento, tais como a segurança, privacidade, heterogeneidade, interoperabilidade e conectividade. Cada vez mais, estes dispositivos necessitam de trocar um elevado número de dados com a Internet, o que afeta severamente o seu desempenho. Além disto, inúmeras aplicações apresentam requisitos tempo-real. Tipicamente, um Sistema Operativo (SO) facilita a gestão destes requisitos. No entanto, SOs de propósito geral não são apropriados para dispositivos *low-end*, devido à limitação de recursos que estes apresentam. Por esta razão, SOs embebidos emergiram como uma solução que combina conectividade entre dispositivos e a Internet, com um impacto reduzido de memória.

SOs embebidos com ligação à Internet frequentemente tem uma pilha de rede integrada, que fornece funcionalidades de conectividade e interoperabilidade. Esta permite uma comunicação estruturada e regularizada, entre dispositivos, recorrendo à sua estrutura de camadas sequenciais, e implementando um conjunto de protocolos e *standards*. Devido às inúmeras pilhas de rede disponíveis, por exemplo, uIP e OpenWSN, os requisitos de interoperabilidade nem sempre são cumpridos. Isto acontece devido ao facto de que cada pilha de rede implementa a sua própria versão de um protocolo, ou adopta standards diferentes em cada camada.

Com o intuito de mititgar estes problemas e visto que não há uma solução generalizada, esta tese focar-se-á no estudo de um SO para IoT e a sua pilha de rede. Irá centrar-se no desenvolvimento de um conjunto de *Application Programming Interfaces (APIs)* agnósticas, com a finalidade de facilitar a integração do SO com diferentes implementações de pilhas de rede. Realizar-se-á com a finalidade de promover os requisitos de interoperabilidade anteriormente mencionados, sem sacrificar o normal desempenho de um nodo, mantendo total conectividade com a rede IoT.

Palavras-chave: Agnosticismo, Dispositivos Low-end, IoT, Pilha de Rede, SO

Abstract

Connecting a multitude of low-end devices in the Internet of Things (IoT) leads to many issues that must be tackled from the early stages of development, such as security, privacy, heterogeneity, interoperability, and connectivity. With the ever-growing amounts of data transferred over the network, performance is a topic that has gained an increased importance. Besides these requirements, several applications also require real-time capabilities, in order to deal with the critical timing constraints. Typically, an Operating System (OS) facilitates managing these requirements. However, traditional general-purpose OSes are not suitable for low-end devices due to their constrained resources. Thus, embedded OSes with small memory footprints emerged as a great solution that combine connectivity between devices and the Internet.

Embedded OSes in the IoT often integrate a network stack, enabling both connectivity and interoperability features. Such network stack enables a structured and standardized communication between devices resorting to its back-to-back layer structure, which implements a set of well-known protocols and standards. Due to the many network stacks available, e.g., uIP and OpenWSN, the interoperability requirements are not always met. This is due to each network stack deploying their own protocol version, or adopting different standards in each layer.

In order to mitigate these problems, and since there is no "one size fits all" solution, this thesis targets the study of a well-known OS for IoT and its network stack, focusing in creating an agnostic Application Programming Interface (API) that can facilitate its integration with other network layers from different stack implementations. This will promote the interoperability requirements without sacrificing the normal behavior of a node, while keeping the full connectivity with the IoT network.

Keywords: Agnosticism, IoT, Low-end Devices, Network Stack, OS.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

6LoWPAN IPv6 over Low Power Wireless Personal Area Networks.

6TiSCH IPv6 over the TSCH mode of IEEE 802.15.4e.

ACK Acknowledgement.

ADC Analog-to-Digital Converter.

AI Artificial Intelligence.

ALS Ambient Light Sensor.

ALU Arithmetic Logic Unit.

API Application Programming Interface.

ARM Advanced Reduced Instruction Set Computer Machine.

CoAP Constrained Application Protocol.

CORE Constrained RESTful Environments.

COTS Commercial Off-The-Shelf.

CPU Central Processing Unit.

DAC Digital-to-Analog Converter.

DAO Destination Advertisement Object.

DODAG Destination Oriented Directed Acyclic Graph.

EB Evaluation Board.

EM Evaluation Module.

GPU Graphical Processing Unit.

HTTP Hypertext Transfer Protocol.

ICMP Internet Control Message Protocol.

ID Identifier.

IDE Integrated Development Environment.

IEEE Institute of Electrical and Electronics Engineers.

IETF Internet Engineering Task Force.

IoT Internet of Things.

IP Internet Protocol.

IPC Inter Process Communication.

IPv4 Internet Protocol version 4.

IPv6 Internet Protocol version 6.

ISM Industrial, Scientific, and Medical.

ISR Interrupt Service Routine.

I²C Inter-Integrated Circuit.

kB Kilobytes.

kbps kilobit per second.

LCD Liquid Crystal Display.

LED Light Emitting Diode.

LLN Low Power and Lossy Network.

LR-WPAN Low-Rate Wireless Personal Area Network.

MAC Medium Access Control.

MB Megabyte.

MCU Microcontroller Unit.

MP2P Multipoint-to-Point.

MTU Maximum Transmission Unit.

op-amp Operational Amplifier.

OS Operating System.

OSI Open System Interconnection.

OTA Over the Air.

P2MP Point-to-Multipoint.

P2P Point-to-Point.

PAN Personal Area Network.

PHY Physical.

RAM Random Acess Memory.

RDC Radio Duty Cycling.

ROLL Routing Over Low Power and Lossy Network.

ROM Read Only Memory.

RPL Routing Protocol for Low Power and Lossy Network.

RTOS Real-time Operating System.

SD Secure Digital.

SFD Start of Frame Delimiter.

SO Sistema Operativo.

SoC System on Chip.

SPI Serial Peripheral Interface.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TSCH Time Synchronized Channel Hopping.

UART Universal Asynchronous Receiver-Transmitter.

UDP User Datagram Protocol.

USB Universal Serial Bus.

WPAN Wireless Personal Area Network.

WSN Wireless Sensor Network.

1. Introduction

The current chapter outlines the planned work for this thesis, exploring the aim of it as well as the several goals achieved to accomplish it. Lastly, the document structure is presented.

1.1 Problem Statement

With the technological evolution seen in recent years, a growing percentage of ubiquitous services and devices ceased to be mostly mechanical and started to incorporate computational power [1]. Furthermore, there is an increasing demand for connectivity and interoperability between embedded devices [2]. The number of untethered embedded devices is increasing, with connectivity becoming a dominant requirement. The Internet being the most prominent network currently existing, it is only logical to connect devices to it, taking advantage of its links and "unlimited vastness". These events led to the rise of the Internet of Things (IoT).

IoT is based on the principle of everything being connected, anytime and anywhere. It provides interaction among the real/physical and the digital/virtual worlds. "Things" (end devices) became context-aware, and they can exchange data and information [3]. These systems are becoming a huge trend and have more impact on our daily routines, such as wearables, smart-locks, smart-scales, etc. [4, 5, 6].

The connection of endless devices to the IoT highlights several challenges in design and deployment. Besides the connectivity and interoperability of heterogeneous wireless nodes, security and privacy are also issues encountered even at the network edge. These issues demand a robust solution since there is an ever-growing amount of private and sensible data transferred over the network. Numerous IoT applications for low-end devices demand real-time capabilities to handle strict timing constraints (automobiles, sensors, monitoring systems, etc.). Also, IoT applications require a standard network stack to keep the interoperability requirements while

handling traffic flow. Typically, the mentioned requirements are supported by the adoption of an embedded Operating System (OS). However, due to low-end devices and their resource-constrained characteristics, general-purpose OSes such as Linux are not the most suitable for them. Currently, there is a vast spectrum of embedded OSes, for example, Contiki, RIOT, mbed, Amazon FreeRTOS, etc. These were conceived to fit the tight constraints of low-end devices. The connectivity and interoperability features are provided by a standard and software-based network stack, which the OS follows strictly.

The problem surfaces as each OS may deploy different implementations of a network stack, which may create interoperability or performance issues. The inclusion or not of specific individual layers (such as energy efficiency-related layers, security-related layers, etc.) creates divergence in the structure of network stacks. Besides, for each layer, a wide range of available protocols may be deployed, in which, for the same protocol, differences in versions and standards may occur. The discrepancy in network stack's implementations leads to performance-related issues, and more significantly, it leads to interoperability issues as the network stack's structures used to process the data exchange in the network dramatically vary.

1.2 Aim and Scope

This thesis aims to create a set of agnostic Application Programming Interfaces (APIs), referred to as the Agnostic Layer, intended to tackle the performance and interoperability problems derived from the multitude of different network stack implementations.

This thesis's main work subjects are a popular OS for IoT, the Contiki OS, and the default network stack, uIP network stack. The main idea is to mitigate the dependency between these two elements by creating an agnostic API. This API will enable integration with other network stacks without sacrificing its normal behavior while providing full connectivity to the IoT network. In order to successfully achieve the primary goal of this work, the following tasks were identified:

- Study the existing IoT OSes for low-end embedded systems (e.g., RIOT, Contiki, mBed, etc.);
- Thorough analysis about the IoT network stack (uIP, OpenWSN, etc.) adopted by each of them;

- Development of an agnostic implementation (by using APIs) to provide independence between OS and network stack.
- Evaluate the performance of the system through several experimental processes.

1.3 Document Structure

This thesis document is structured as follows:

- **Chapter 1, Introduction:** provides a small introduction to the thesis, the problem that it intends to tackle as well as its aim and scope.
- **Chapter 2, State of the Art:** provides a background of the mechanisms and protocols used throughout this work. The IoT is briefly introduced, followed by a review of the OSes for IoT, focusing on the types of devices that support them and the challenges currently faced. Afterwards, some design choices are analyzed, followed by a brief dissection of some open-source OSes for IoT. Lastly, the network stack is reviewed as well as all protocols used throughout this thesis, providing examples of appropriate network stacks.
- **Chapter 3, Platform and Tools:** focuses on the software (all programs and extensions) and hardware (microcontroller boards and its evaluation modules) used to assist the development and testing of the thesis.
- **Chapter 4, Design:** presents the theoretical aspects relevant for this thesis. The design challenges found throughout this work are analyzed and explained, as well as the solution employed to tackle them. There is also a comprehensive explanation of what agnosticism is and how it is a goal in this project. The last topic of this chapter is the *raison d'être* of this thesis: The Agnostic Layer, its design, its positioning in the whole system, and some key points about it.
- **Chapter 5, Implementation:** mainly focuses on the practical aspects of the Chapter 4. The chapter presents the implementation of the Agnostic layer and the solutions to the problems that emerged from it. It also provides insights, code-wise, about the network stack's selection, Agnostic Layer's configuration, and pinpoints relevant functions to it.

- **Chapter 6, Evaluation and Results:** highlights the results taken from the implementation of the previous Chapter and how it affects the system. A results' analysis is made comparing the system's performance when integrating the Agnostic Layer with its baseline values to fully understand the impact the latter has on the system.

- **Chapter 7, Conlusion:** concludes this work and takes a look into the horizon with future work to be done.

2. State of the Art

This Chapter contemplates an introduction, description, and proper understanding of the underlying protocols and mechanisms which revolve around this thesis.

Section 2.1 presents a brief introduction to the IoT, an example of an IoT system, and some background, more specifically, some historical events that shaped the IoT world. Section 2.2 details the type of devices that compose an IoT network, the challenges, and design choices for an IoT OS are described, and some open-source OSes are presented. The final section of this Chapter, Section 2.4, comprehends a more complex and in-depth analysis of a network stack and its several features. It details some popular protocols that are usually adopted in each layer of the network stack. A brief description of two important network stacks (uIP network stack and OpenWSN network stack) is presented in the end.

2.1 Internet of Things

The Internet has become a significant network infrastructure that keeps connecting myriads of endpoint devices, such as desktop computers, servers, embedded devices, etc.

Defined as “the network of things”, IoT incorporates embedded electronics with everyday physical objects [3], facilitating the gathering and exchanging data for several services and purposes [7]. IoT is present in several ecossystems like company process automation, aimed at reducing costs; optimizing home heating systems, with the objective of minimizing power consumption, resulting in a smaller environmental impact [5]; monitoring platforms, such as recycle bins [6]; lighting and traffic control as well as pollution and energy consumption control in smart cities [8]; highways monitoring systems with collisions detection [4], and many others. Figure 2.1, adapted from [9], exemplifies a possible IoT system. Sensors, antennas, and data

collected by other systems (such as microcontrollers), are transmitted to a hub, which groups different data together, and through the analysis of that information, an action is performed.

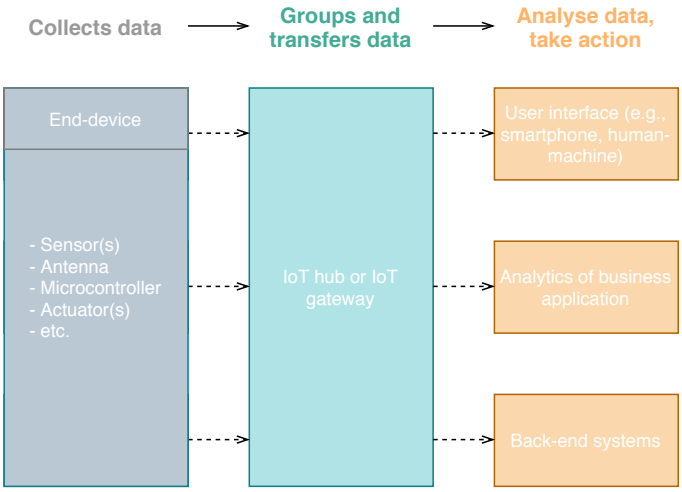


Figure 2.1: IoT system.

Although the IoT concept being extensively used, its definition is still blurry due to its broad application in different domains. IoT requires a software and hardware architecture that must be able to deal with the large amount of data that is continually being generated. These architectures must be able to deal with data processing paradigms, data mining, stream processing [2], Artificial Intelligence (AI), data filtering, etc. On top of this, an IoT device must comply with high energy optimizations techniques due to high power consumption restrictions since batteries typically power these devices. Table 2.1, based on [10], shows some events that helped shape the IoT history, from its early beginnings to modern developments. Of the dates presented, some were highly important events that are stamped on IoT history by significantly improving it and vastly propelling the technology, such as 1998's introduction of Internet Protocol version 6 (IPv6) (enhancing performance, security, and allowing a wider range of Internet Protocol (IP) addresses) or 2004's IoT tittle first appearances.

The different IoT use cases presented throughout this section require the deployment of different software to model the behavior of these systems for the intended applications. In addition to the software deployment, the hardware itself also varies greatly, as different applications require different platforms to operate, which can be high-resource or low-resource requiring ones.

Table 2.1: Events that shaped the world of IoT history.

Year	Event
1982	Carnegie Mellon University students invent ARPANET-connected coke machine
1989	Tim Berners-Lee proposes the World Wide Web
1990	John Romkey introduces a toasted connected to the Internet
1994	Steve Mann creates a wearable Internet-connected camera
1998	Introduction of IPv6
1999	Kevin Ashton introduces the term "Internet of Things"
2000	LG announces the world's first Internet-enabled refrigerator
2004	The term "Internet of Things" first appears in book titles
2008	The first International Conference on the IoT
2010	China picks IoT as key industry to tackle financial problems
2014	The number of devices exceeds the number of people
2017	Governments start to think about IoT security; Cryptocurrencies focus on IoT

2.2 Devices for the Internet of Things

Due to its nature, the IoT is not tailored to have every system with the same hardware and software architectures. The end node devices that compose the IoT assume a multitude of shapes created with a plenitude of different purposes, such as wearables, smart-locks, smart-scales, etc. [4, 5, 6]. The heterogeneity in IoT is even more distinguished than in the traditional Internet due to the myriad of different end node devices connected to it [2].

Based on their capacity and performance, Hahm *et al.* classify the IoT end node devices into two big categories [11]:

- **High-end IoT devices:** These devices have enough resources to run a traditional general-purpose OS, such as a Linux distribution. They possess high storage resources, powerful Central Processing Unit (CPU) capabilities, possibly a Graphical Processing Unit (GPU), etc.

- **Low-end IoT devices:** These are very constrained devices when it comes to their resources like power availability, memory storage, and CPU processing capabilities. Typically single-core, these resource-constrained devices [12] are not the most suited to run traditional general-purpose OSes.

Furthermore, low-end IoT devices can be sub-categorized into three distinct classes according to their constraints and computational power. Table 2.2, as seen on [13], classifies these devices according to their required memory resources.

Table 2.2: Classification of IoT devices.

Name	Abbreviation	Data Size (e.g. RAM)	Code Size (e.g. Flash)
Class 0	C0	<1KiB	<100KiB
Class 1	C1	Approx. 10KiB	Approx. 100KiB
Class 2	C2	Approx. 50KiB	Approx. 250KiB

Class 0 devices are highly constrained, both in terms of memory and processing capabilities, to the point that not all can communicate with the Internet. Communication with the Internet will happen with the help of other devices, acting as proxies, gateways, or servers. Rarely reconfigured, Class 0 devices are typically preconfigured [13]. High specialization and resource limitations make the use of a traditional general-purpose OS very unsuitable. Therefore, the software used on these devices is very hardware-oriented and typically developed bare metal [11].

Class 1 devices are somewhat constrained in memory and processing capabilities. Communication with other devices is possible; however, the usage of complex protocols, such as Hypertext Transfer Protocol (HTTP), Transport Layer Security (TLS), etc., is out of the capability presented by these types of devices. Resorting to protocols specially designed for constrained nodes, such as Constrained Application Protocol (CoAP) over User Datagram Protocol (UDP), Class 1 devices can communicate without the need for gateways or other sophisticated devices. These devices can provide support for security functions on a vast network [13].

Class 2 devices are less resource-constrained, supporting more "heavy" protocols. Regardless of their computational power, these devices still benefit from lightweight and energy-efficient protocols [13]. It is foreseen that IoT devices will be cheaper in the future. Contrary to Moore's Law [1], IoT devices will not tend to have more memory or performance but will become cheaper and smaller [13].

[1] Refers to Moore's idea that the number of transistors on a chip doubles every two years. The speed and performance of chips also increases along with the number of transistors added.

Low-end IoT devices with capabilities higher to Class 2 devices do exist. However, the Internet Engineering Task Force (IETF) working group "does not make an attempt" to define them [13].

As seen throughout the current section, low-end IoT devices can be highly divided according to their hardware capabilities, which greatly vary between different types of devices. Besides that, high-end IoT devices also present different hardware characteristics that differentiate between each other. However, these types of devices are out of the scope of this thesis. In conclusion, the heterogeneity in the IoT is enormous, which hinders the software development task for millions of different hardware types.

An OS typically implements APIs whose primary purpose is to abstract the software from the hardware. These APIs allow hardware-independent code production and large-scale software development.

2.3 Operating Systems for the Internet of Things

OSes for the IoT comprise a vast number of characteristics that specifically identify them. The purpose of the current section is to highlight a collection of requirements that IoT OSes must comply with and design choices that model the application, allowing specific features in exchange for others (a trade-off). At the end of the section, open source OSes are discussed, as well as their features.

2.3.1 Challenges for an IoT Operating System

The following section presents diverse requirements associated with the IoT that are imposed on the OSes that run on low-end devices. These requirements highly impact the OS behavior and suitability to be used by given hardware.

2.3.1.1 Energy Efficiency

Typically, low-end IoT devices, such as sensors, run on batteries. In some cases, the deployed devices are in harsh environments. Usually, battery exchange is an arduous task to execute thus, energy efficiency is a highly required feature [14].

Some IoT systems resort to software in order to maximize its hardware batteries' life. Techniques such as Radio Duty Cycling (RDC) (that works by turning the radio on sleep mode as frequently as feasible) are crucial to maintaining the systems running for as long as possible, avoiding constant batteries' replacements.

2.3.1.2 Memory Footprint

Low-end IoT devices are resource-constrained machines, especially in terms of memory, therefore memory management is a critical function of the OS [15, 16]. While high-end IoT devices provide high amounts of memory, low-end IoT devices may provide as low as less than 10 kilobytes of Random Acess Memory (RAM) and less than 100 kilobytes of Read Only Memory (ROM) [11]. Another requirement is having an optimized set of APIs as well as data structures to comply with memory footprint constraints, in which these OSes must fit into.

2.3.1.3 Network Connectivity

Each IoT device has a network interface that can vary from device to device [17]. These types of devices interconnect and communicate with endless devices. Therefore, it is expected that IoT devices seamlessly communicate with each other [18].

In a tremendously extensive network, it is mandatory that IoT systems support IPv6 to have unique identities, to be able to communicate with other Internet hosts. This requirement, altogether with having to support multiple link-layer technologies [11], created the path to IP protocols-based network stacks. Ideally, the IoT OS should cater to multiple network stacks [11, 15]. Therefore, software-wise, in order to provide support for wireless interfaces, an embedded network stack must be deployed [19].

2.3.1.4 Security

IoT systems are usually part of critical infrastructures, some with life safety implications. Besides that, some IoT devices are connected to the Internet. Thus, a high level of security and privacy standards is paramount [20].

An IoT OS should provide adequate mechanisms to enforce security requirements. These mechanisms must include and provide control over some challenges such as authentication

by verifying the identity of devices; access control by allowing or preventing a given node from accessing guarded data/information; and data integrity by maintaining the validity and accuracy of data [11].

2.3.1.5 Support for Heterogeneous Hardware

There is a multitude of different IoT nodes currently active, which incorporate various hardware. They differ in many characteristics including RAM, ROM, memory structure, connectivity technologies, etc. The architectures on these systems also vary greatly. IoT devices are based on 8 bit, 16 bit, 32 bit, and 64 bit architectures [11]. A generic OS for low-end IoT devices must support the immense heterogeneity in hardware and communication technologies present in today's IoT world.

2.3.2 Operating System Design Choices

Since there is a growing focus on IoT and its use cases, a myriad of OSes entered the scene, providing a broad array of design choices [21]. Every OS has specific design architectures (they have their particular characteristics, allowing specific capabilities and flexibilities). A trade-off has to be made when considering the OS choice, upon application constraints. A possible scenario would be the balance between performance and energy usage. If computational power arises, so does the energy required. Many features can be analyzed, such as memory allocation, network buffer management, etc.; however, the ones that can highly differentiate the several OSes are the following: architecture, programming model, and scheduling.

2.3.2.1 Architecture

The OS architecture resides in the choice of the kernel type. There are different kernel types, and they vary in many traits such as complexity, security, flexibility, robustness, etc. These are:

- **Monolithic Kernel:** The approach on these types of kernels is oriented to a more straightforward design, in which the components of the kernel are all developed together [11]. Consequently, the monolithic kernel itself becomes very large. Because of these features, monolithic kernels are efficient, although providing little flexibility, portability, and

requiring much memory [22]. Without a defined structure, the maintenance and future changes of a monolithic kernel affect other parts of it [23].

- **Microkernel:** The microkernels' approach is based on the development of only essential core OS functions in the kernel, while other functionalities needed are inserted as modules [23]. The modules are inserted in the user address space as processes. This creates a layer between the user address space and the kernel address space, denying software applications direct communication with the kernel [22]. The advantages around these types of kernels are their high portability, flexibility, and robustness, although being more complex systems [11, 22]. This kernel type is robust against bugs that happen in a single component [24]. This feature means that if a given component fails, for some reason, this failure does not compromise the rest of the system.

- **Hybrid (or Modular) Kernel:** This kernel type is a mix between the monolithic kernel and the micro kernel. Basically, a monolithic kernel with the majority of device drivers removed. Device drivers are attached to the kernel as required when starting up or running. To reduce the performance overhead imposed of the traditional micro kernel, the hybrid kernel has some services in the kernel address space, and kernel code in the user address space [25].

2.3.2.2 Programming Model

The programming model has high importance in how the program can be modeled and in application development. In the OS for IoT arena, programming models can be mainly divided into multithreaded programming and event-driven programming.

Multithreading programming is the standard approach for traditional and modern OSes [11] that allows tasks to run, each one in its context, managing its own stack, and communicating between themselves using Inter Process Communication (IPC) APIs. The CPU switches execution between the different threads, culminating in concurrent execution [26]. However, they are resource-intensive, especially in terms of memory overhead [11].

Event-driven programming is generally used for Wireless Sensor Networks (WSNs), which have scarce resources. A task is activated by an event, generally external, signaled by an interrupt. Useful for scarce-resource devices due to its low complexity and memory consumption [11, 27].

However, this programming model expresses the program as a finite state machine that is not readily applicable to all programs [28].

2.3.2.3 Scheduling

Scheduling algorithms, performed by a scheduler, determines which task should execute, in a given moment, in the CPU [27]. The scheduler provides real-time capabilities, and affects energy efficiency and resource usage [29]. The scheduler can be classified as preemptive or cooperative (or others, but for the purview of this thesis, these are ignored). A preemptive scheduler has the capability to interrupt any task to allow other tasks execution time [11]. The cooperative scheduler does not allow task interruptions by other tasks (and sometimes not even the kernel itself). As stated by Hahm *et al.* [11], using a preemptive scheduler may have implications in an IoT system. This happens due to the need to have at least one hardware timer active to time-slice each task CPU's access. The need to have this component of hardware active all the time prevents the device from entering the deepest power-saving mode.

2.3.3 Open Source Operating Systems

Although there are many proprietary OSes, the focus throughout this thesis relies on open source OSes. There are many advantages to using an open source OS, such as availability, greater security [30], extensive customization, etc.

Table 2.3, adapted from [11], presents an overview of different open source OSes (emphasis on the design choices previously mentioned in section 2.3.2). It is worth noting that the majority of the OSes presented do not have a native network stack. They either do not have a network stack or resort to external network stacks, with Nut/OS and nuttX being the exceptions. Besides the design choices that were previously analyzed, one other aspect that should be emphasized is the programming language the OSes use. They all use the C programming language or a variation of it, such as C++ or nesC.

The goal in the current section is an overview of OSes. Contiki and RIOT are open source OSes gaining significant prominence in the low-end IoT "world". Therefore, these two OSes are dissected in the following sections.

Table 2.3: Overview of different Open Source OSes.

Name	Architecture	Scheduler	Programming Model	Targeted Device Class	Programming Languages	Network Stacks
Contiki	Monolithic	Cooperative	Event-driven, Protothreads	Class 0 +1	C	uIP, RIME
RIOT	Micro Kernel	Preemptive	Multithreading	Class 1 + 2	C, C++	gnrc, OpenWSN, ccn-lite
FreeRTOS	Micro Kernel	Preemptive/Cooperative	Multithreading	Class 1 + 2	C, C++	None
TinyOS	Monolithic	Cooperative	Event-driven	Class 0	nesC	BLIP
OpenWSN	Monolithic	Cooperative	Event-driven	Class 0 + 1 + 2	C	OpenWSN
nuttX	Monolithic or Micro Kernel	Preemptive	Multithreading	Class 1 + 2	C	Native
eCos	Monolithic	Preemptive	Multithreading	Class 1 + 2	C	lwIP, BSD
uClinux	Monolithic	Preemptive	Multithreading	>Class 2	C	Linux
ChibiOS/RT	Micro Kernel	Preemptive	Multithreading	Class 1 + 2	C	None
CoOS	Micro Kernel	Preemptive	Multithreading	Class 2	C	None
nanoRK	Monolithic (resource kernel)	Preemptive	Multithreading	Class 0	C	None
Nut/OS	Monolithic	Cooperative	Multithreading	Class 0 + 1	C	Native

2.3.3.1 Contiki

Initially developed for WSN sensor nodes, Contiki is now a lightweight and flexible OS for IoT. It is highly portable and built around an event-driven/protothreads kernel. A typical Contiki configuration requires around 2 Kilobytes (kB) of RAM and 40kB of ROM [27]. Figure 2.2, adapted from [31], represents the Contiki OS architecture. The different layers that compose the OS are presented in the figure. Also, as seen in the Figure, each hardware module communicates directly with their correspondent drivers, as well as the driver's layer that communicates directly with the Contiki core layer. This last layer encompasses the uIP network stack, which became integrated with Contiki OS by default since version 0.9. It also includes the library for protothreads, which are lightweight stackless threads characteristic of Contiki.

Contiki implements a cooperative scheduler. It also supports dynamic memory management as well as dynamic linking of the programs. To protect the system from memory fragmentation, it uses a Managed Memory Allocator [32], which compacts the memory when blocks are free.

Contiki uses extremely lightweight stackless threads, intended for WSNs or severely memory-constrained systems, called protothreads. Protothreads provide linear code execution for event-driven systems [33]. Applications can use both versions of IP; this is Internet Protocol version 4 (IPv4) and IPv6. It integrates the uIP network stack, which is an IoT-compliant and standardized communications network stack [34]. Contiki does not support real-time applications since it does not implement any real-time processing scheduling algorithms, nor does it provide support for secure communications. Contiki implements a file system support for flash-based sensor devices called the Coffe File System [35]. Finally, Contiki provides network simulations trough the Cooja simulator [36].

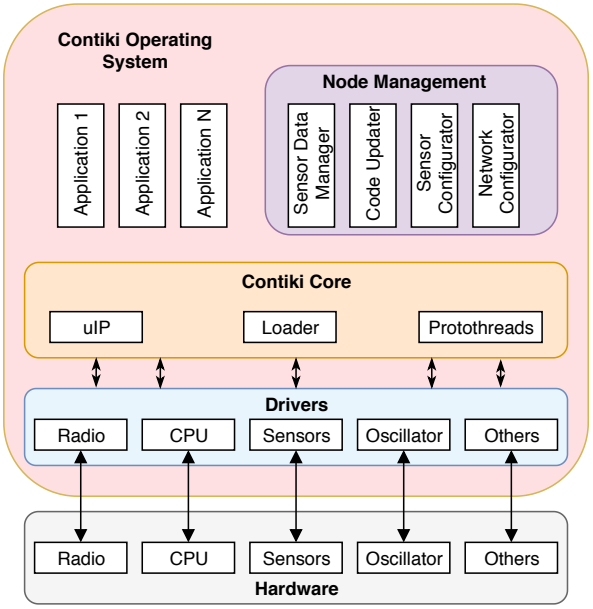


Figure 2.2: Contiki OS Architecture.

2.3.3.2 RIOT

RIOT is based on a micro kernel architecture [37], with a configuration of just 1.5kB of RAM and 5kB of ROM. Figure 2.3, adapted from [38], represents the RIOT OS. It is worth noting the similarities it has with the Contiki OS. Its lowest layer is the hardware, followed by the drivers (hardware abstraction) and the kernel. The upper layers are the libraries of the system and the embedded IP network stack. On top of all are inserted the applications that run on the OS.

RIOT implements a preemptive, priority-based scheduler and provides support for IPC and multiple task priorities [29]. It supports real-time scheduling due to its programming model. It also supports both static and dynamic memory allocation [39]. When it comes to the network

stack, RIOT supports IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN), Routing Protocol for Low Power and Lossy Network (RPL) (non-storing and storing mode) and provides full support for IPv6, UDP, and Transmission Control Protocol (TCP) [29].

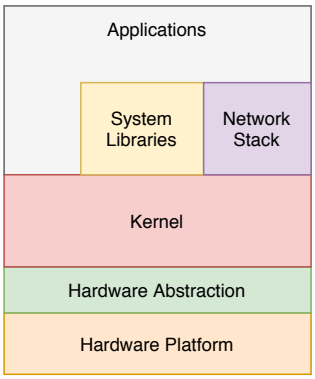


Figure 2.3: RIOT OS Architecture.

2.4 Network Stack

Open System Interconnection (OSI) Reference Model [40] defines a seven-layer model, created in 1979 mainly to address the heterogeneous network interconnection compatibility issues, that was intended to serve as a form of framework for the definition of standard protocols [41, 42]. Figure 2.4 shows the OSI seven-layer network model, formed by the layers: physical, data link, network, transport, session, presentation, and application. The OSI model was a complete and robust network architecture. However, it was challenging to adapt and change, so it lost influence to the TCP/IP model [42].

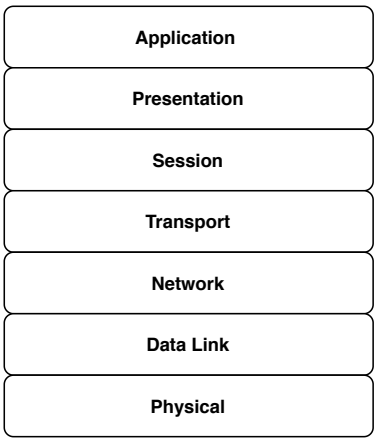


Figure 2.4: OSI model network stack layers.

Universally adopted, TCP/IP is the standard protocol for wired networking [43]. Due to extreme communication conditions, restraints, and specific requirements, TCP/IP protocol stack is often labeled as unsuited for IoT nodes. According to Dunkels *et al.* [44], five constraints severely prevent the TCP/IP protocol stack from being IoT's network stack. These are:

- **IP addressing architecture:** In typical IP networks, address assignment is performed by manual configuration or by a dynamic mechanism. On a large scale, manual configuration is not possible, and dynamic methods are resource-draining.
- **Header overhead:** TCP/IP headers are very large, which imposes a significant header overhead.
- **Address-centric routing:** Data-centric routing mechanisms are preferable over TCP/IP's address-centric mechanisms due to their redundancy of data traffic [44, 45].
- **Limited resources:** IoT devices are generally limited in terms of memory and processing power. TCP/IP protocol network stack is too heavy for these devices.
- **TCP performance and energy inefficiency:** The TCP protocol is very reliable due to its many retransmissions and end-to-end acknowledgment, and that imposes performance issues in wireless networks.

In order to tackle the problem of lack of standards and scarcity of low power and low bandwidth protocols, the Institute of Electrical and Electronics Engineers (IEEE) and IETF bodies started, in 2003, to create a framework for communication protocols oriented to low-end IoT nodes, such as RPL and CoAP, amongst others [46]. From this point on to the end of the chapter, *de facto* standards regarding layers for network stacks for low-end IoT devices are explored, as well as their interactions with each other.

2.4.1 Network Stack Layers

Each network stack has its very own and unique set of characteristics. Due to the myriad of protocols currently available, a network stack can present different behaviors by modulating these parameters. On top of this, a network stack can be modular or tightly interconnected with the OS it functions with. Besides that, if a network stack is interconnected with the OS, there can be

several degrees of dependency. Furthermore, many more variables are present when selecting a network stack. So, it can be inferred that categorizing a network stack is not a simple task.

Protocols for a network stack are presented next. For the sake of straightforwardness, a generic network stack is assumed. Besides its protocols, all other features are obsolete, and the protocols themselves are the most notorious ones - the ones more heavily used and explored in the literature. Figure 2.5 represents all the generic network stack layers that are used to illustrate the several protocols and their disposition within a network stack. For each of the layers represented in the figure, a protocol is associated and explained in the next sections.

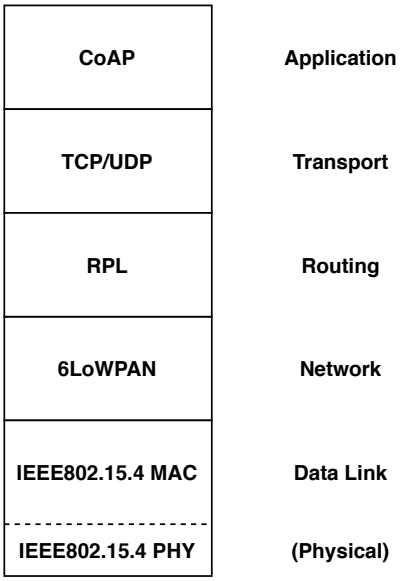


Figure 2.5: Typical network stack layers.

2.4.1.1 IEEE 802.15.4 PHY

The most prominent standard in low-power radio technology [47], IEEE 802.15.4, is a radio technology standard for low-end IoT devices [46]. It specifies both Physical (PHY) and Medium Access Control (MAC) layers. This standard was and has been developed and improved by the IEEE 802.15 Personal Area Network (PAN) group.

The IEEE 802.15.4 PHY layer typically runs in the 2.4GHz - 2.485GHz frequency band, which is an unlicensed frequency band. It defines 16 frequency channels, each one separated by 5MHz each, between 2.405GHz and 2.480GHz. Each channel size is 2MHz wide, which prevents channel communications from interfering with one another. The radio transceiver can freely send and receive on any channel, with a switching time of no more than 192 μs [2].

When sending a packet, after the Start of Frame Delimiter (SFD) transmission, which indicates the start of the payload transmission, the very first byte of the payload pinpoints its length. If a radio is "listening" when no other mote is transmitting, it "hears" white/random noise. The data payload's maximum value is 127 bytes, limiting the maximum length of a packet to 128 bytes, considering the first byte. After receiving the payload, the radio loads a buffer with the number of bytes, pinpointed by the length byte, with the received data. After this operation is concluded, the radio transceiver indicates a successful reception to the controller system [2].

2.4.1.2 IEEE 802.15.4 MAC

IEEE 802.15.4 defines the format of the MAC header and how motes can communicate with each other. According to Palattella *et al* [2], this protocol is ill-fitted for low-power multi-hop networking due to: powered routers, since using IEEE 802.15.4 MAC for multi-hop routing forces the radio transceiver to be active all time, severely draining batteries; and single-channel operation, because this may cause shadowing and multi-path fading, causing network instabilities or even network collapse.

Since on radio, the idle mode consumes much less energy than in listen or transmit modes; a radio power management mechanism has to be implemented, such as RDC, to achieve power savings. Therefore power management mechanisms became an essential part of the MAC layer [48].

These problems previously mentioned led to the creation and implementation of Time Synchronized Channel Hopping (TSCH), which became part of the MAC protocol since 2010 and, according to Palattella *et al.* [2], TSCH is the latest generation of highly reliable and low-power MAC protocol. By using the mechanisms of time synchronization and channel hopping, TSCH guarantees high reliability and low duty cycles, resulting in the utmost power efficiency [49]. Next, a list of features that compose TSCH is presented and explained based on Palattella *et al.* [2].

Slot Frame Structure A slot frame is a set of slots that repeats itself over a period of time (cycles). Following a scheduling mechanism, each mote either transmits, receives, or sleeps (radio off), according to its instructions. At each transmission slot, if the MAC layer has a packet (generated by an upper layer) to a particular neighbor associated with that slot, it transmits the packet and waits for the Acknowledgement (ACK). Otherwise, it goes back to sleep (turns the

radio off). If no ACK is received, the MAC layer keeps the packet in transmission buffer for future re-transmission. At each reception time slot, the mote turns its radio on moments before the expected time to receive the packet. If the reception is successful, an ACK is sent to the transmitter, the radio is turned off, and the received packet is sent to upper layers for processing. If no packet is received after a delimited time interval, the radio is turned off. Figure 2.6, based on [2], represents a slot frame topology. In this figure, the slot frame is composed of three time slots, numbered from zero to two, that repeat themselves over a period of time, constituting a cycle. Each time slot can be a reception or transmission slot, although that is not identified in the figure.

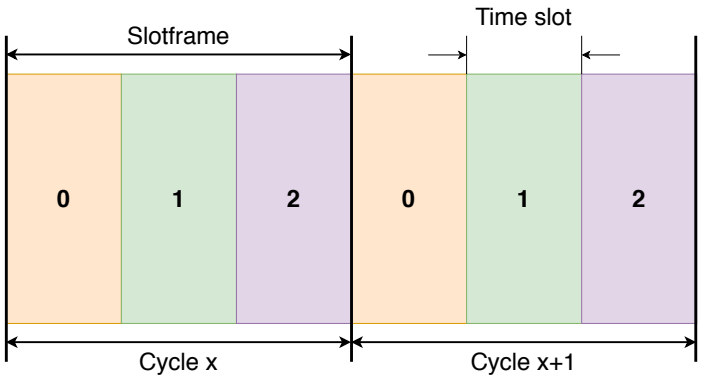


Figure 2.6: TSCH slotframe

Scheduling A scheduling mechanism must ensure that when a given mote wants to transmit to another mote(s), the corresponding mote(s) must be transmitting and receiving accordingly. Likewise, if a mote is dropped from the network, the other motes can not communicate with the removed mote. Scheduling can be performed in a distributed or centralized way. The centralized approach revolves around a specific mote creating and managing the network schedule and instructing the other motes about their links. The distributed approach is based on motes linking with local neighbors.

Synchronization There are two existing approaches for mote synchronization to the network: *Acknowledgment-Based Synchronization* revolves around the receiver calculating the difference between estimated frame arrival time and real frame arrival time and sending this difference to the transmitter so it can synchronize its clock. *Frame-Based Synchronization* revolves

around the receiver calculating the difference between estimated frame arrival time and real frame arrival time and synchronizing its clock.

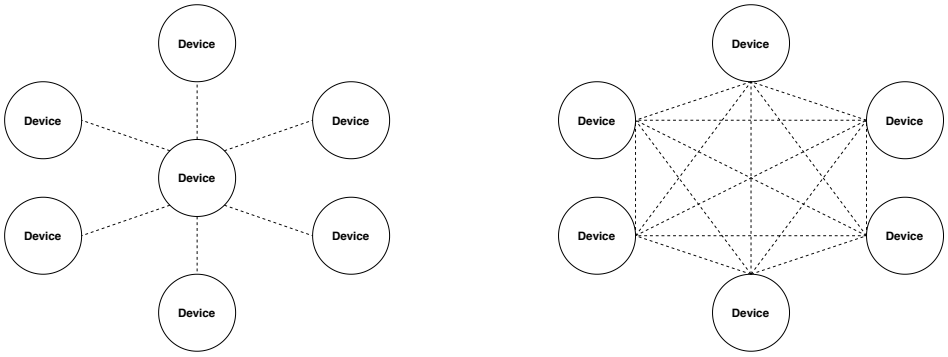
Channel Hopping For every scheduled slot, the scheduler assigns a slot offset and a channel offset. Typically, TSCH uses 16 numbered channels for communication. This number is denominated channel offset. When a node, for example, A has a transmit time slot, transmitting to node B on channel offset 5; node B has, antagonistically, a receive time slot, receiving from node A on the same channel offset.

Network Formation When a new mote tries to join the network, it waits for the reception of an *advertisement* command frame. Upon the reception of this frame, the mote joins the network by sending a *join request* command frame to the advertising device which sent the advertisement frame.

In a distributed scheduling approach, individual motes can process this operation locally. In a centralized scheduling approach, join request command frames are forwarded to the manager mote.

2.4.1.3 6LoWPAN

Usually, low-power Wireless Personal Area Networks (WPANs) are formed by low-cost and battery-powered devices; have unknown positions; are very unreliable; need to stay long periods in idle mode (in which communications are stopped to save energy); and can only transmit packets with small sizes (which have a maximum size of 127 bytes) [2]. These low-power WPANs have low bandwidth and have to provide support for addresses with varying lengths and can also be formed in a star topology, as seen in Figure 2.7a or mesh topology, in Figure 2.7b. With these devices and WPANs characteristics, the implementation of IPv6 is not an easy and straightforward task. However, it is a must since this protocol has essential key features such as universality, extensibility, and stability [48], thus becoming a *de facto* solution. Figure 2.7 represents the models for star topology and mesh topology. In a star topology the end nodes are connected to a centralized one, while in a mesh topology all nodes are connected to each other.



(a) Star Topology Model. **(b)** Mesh Topology Model.

Figure 2.7: Mesh and Star topologies models.

A critical aspect of performing with IPv6 instead of IPv4 is the much more extensive range of addresses. While IPv4 addresses devices with a 32-bit IP address, IPv6 has a 128-bit IP address. Since countless devices are starting to be connected to the IoT, the broader range of addresses is essential.

Starting in 2007, the IETF 6LoWPAN working group started working on specifications and protocol optimization of IPv6 over networks using IEEE 802.15.4. There are some significant challenges when it comes to implementing IPv6 over the IEEE 802.15.4 network. For instance, IPv6's minimum Maximum Transmission Unit (MTU) size (1280 bytes) does not fit into an IEEE 802.15.4 packet frame. Another problem revolves around the significant header overhead by IPv6's header that would waste the very scarce bandwidth of the PHY layer. On top of this, and according to Gomes *et al.* [50], if all devices resort to a unique IPv6 address using a 6LoWPAN network, this would cause a massive overhead that would affect the way small IoT devices perform. To tackle these issues, 6LoWPAN introduces a layer to segment the IPv6 packets into smaller ones required by the lower layers. It also specifies stateless compression to reduce the overhead of IPv6, scheme supporting mesh routing, and simplified IPv6 neighbor discovery protocol [48]. It is worth noting that this protocol is part of the foundation of standardized WSNs, as mentioned in [51]. Due to its important features, such as the ability of devices to perform at efficient and scalable WSNs [52], the capability of permitting ubiquitous connectivity/interoperability between heterogeneous devices, universality, stability, etc. under a lightweight implementation, suited for low-end IoT devices, 6LoWPAN became a standard in Low-Rate Wireless Personal Area Networks (LR-WPANs) [53].

2.4.1.4 RPL

According to the Thubert *et al.* [54], due to the many IoT nodes' constraints, such as loss-prone radio-links, multi-hop mesh topologies with recurrent changes, etc., the IETF Routing Over Low Power and Lossy Network (ROLL) working group has been developing RPL, tackling these routing issues and providing support for a vast range of link layers.

In a typical topology setting, as depicted in Figure 2.8, based on [55], the nodes that compose a network are connected to one or more root devices. These root devices collect data and coordinate motes and link paths. For every single root device, RPL creates a Destination Oriented Directed Acyclic Graph (DODAG). Typically, each node has multiple parents that are closer to the root, although usually only one is used (a preferred one). This communication scheme is called Multipoint-to-Point (MP2P), which supports communication from the nodes to the root, as seen in Figure 2.9a. In this figure, based on [55], the root node is A. Nodes B and C are parent nodes of nodes E and F, respectively. One important aspect is that node D simultaneously has two parent nodes (B and C). In this case, it forwards its data to node B, which is his preferred parent node.

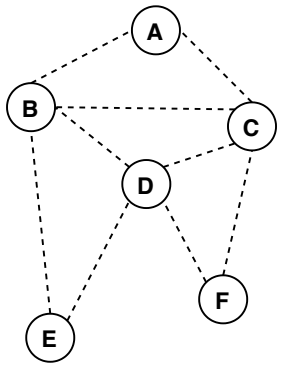


Figure 2.8: RPL routing topology sample wireless network.

The dual of MP2P is called Point-to-Multipoint (P2MP). This communication scheme allows traffic flow from root to nodes, allowing dual traffic orientation. It requires specific control messages called Destination Advertisement Object (DAO) control packets. These messages are broadcasted "upward" in the DODAG topology scheme, via parent nodes, creasing "downward" routes. Figure 2.9b presents this mechanism.

RPL defines two modes of device operation: storing and non-storing. In storing mode, a node maintains a routing table, in which are present mappings of all destinations with regards to next-hop nodes. In non-storing mode, the routing table is kept only to root devices.

When two nodes have to communicate, which is called Point-to-Point (P2P) communication, the two modes of operation exert influence. In storing mode, data packets are forwarded "upward" until a node with routing information is reached (Figure 2.9c). In non-storing mode, data packets are forwarded to the root, which reveals the hops and next-hops packets have to go through in order to reach their respective destination (Figure 2.9d).

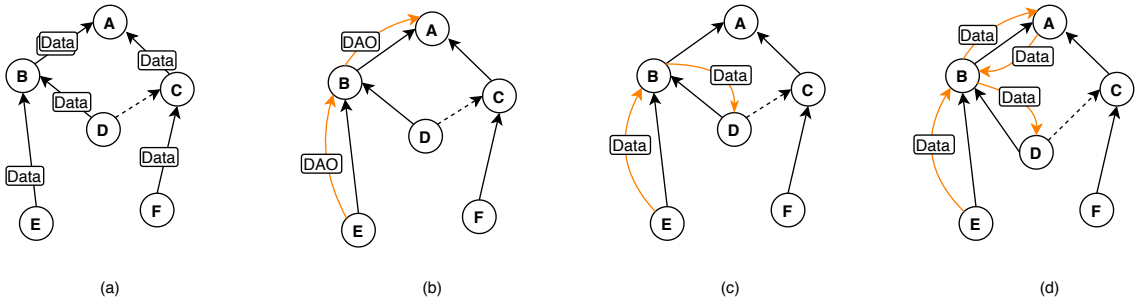


Figure 2.9: Routing in RPL: a) MP2P communication; b) P2MP route construction: storing mode; c) P2P communication: storing mode; d) P2P communication: non-storing mode.

2.4.1.5 TCP and UDP

The network layer, 6LoWPAN, can not guarantee P2P reliability [2]. Typically, this is at the upper-layer's responsibility (transport layer). At the transport layer level, there are two highly used and well-known protocols, TCP and UDP.

TCP provides reliable data transfers between two nodes in the network. A TCP operation is divided into three stages: a connection establishment, data transfer, and connection termination. In order to ensure reliable data transfer, it requires that all transfers from the transmitting node are validated and acknowledged by the receiving node. On top of this, the two nodes must have a connection established, as previously mentioned. Each packet transferred has a sequence number identifying it. If a data packet has been lost or not received for some reason, it retransmits the packet. This mechanism encompasses lost and duplicate packets issues and provides ordered transfers. TCP also guarantees congestion control and traffic control [56]. However, all this control and transfer reliability come at a cost.

TCP imposes a massive overhead to the system for every packet transmitted. Thus, the UDP's usage presents a good alternative for the transport layer in Low Power and Lossy Networks (LLNs) in some cases. UDP, a datagram oriented protocol, transmits its data packets (datagrams) in a best-effort fashion, i.e., no ACK is transmitted. On top of this, the transmission does not require

an established connection from the two endpoints. Loss is tolerable, and in fact, packet loss is expected. Datagrams may be received duplicated and disorderly. However, in real-time systems, where data packet delay may be worst for the system than occasional packet loss, UDP creases its advantages like video streaming and audio streaming. Table 2.4, based on [57] and [58], summarizes the differences between the two protocols.

Table 2.4: Differences between TCP and UDP protocols.

Transmission Control Protocol (TCP)	User Datagram Protocol (UDP)
Connection-oriented protocol. Must establish a connection before data transmission. Must close connection after data transmission.	Datagram-oriented protocol. No connection establishment is needed for transmitting data.
Reliable as it guarantees delivery of data.	Data delivery is not guaranteed.
Ordered. It ensures correct order of data packets.	Not ordered. Data arrives in order of receipt.
Extensive error checking mechanisms.	Basic error checking mechanisms.
Provides congestion control.	No congestion control.
Lower speed.	Higher speed.
No broadcast support.	Broadcast support.
Heavy overhead.	Light overhead.

2.4.1.6 CoAP

According to [59], CoAP is a client-server IoT protocol. This protocol revolves around requests from the client and responses from the server. It is intended for LLNs, at which both endpoint nodes may act as servers and clients. Also, the methods are the similar to HTTP's, which are DELETE, GET, POST, PUT. IETF Constrained RESTful Environments (CORE) working group is responsible for the design and development of CoAP. It has been created, taking into account the restrictions of LLNs, such as reliability and high-congestion environments. CoAP is an asynchronous message exchange protocol that works over a datagram oriented transport layer (e.g., UDP). This happens due to UDP's asynchronous nature.

CoAP is divided into two layers: Messages Layer and Request/Response Layer, as shown in Figure 2.10. The Message layer deals with asynchronous message exchanges over UDP. There are four different message types:

- **Confirmable:** Messages that require a response back.
- **Non-Confirmable:** Messages that do not require a response back.

- **Acknowledgment:** Messages that confirm the reception of a Confirmable message.
- **Reset:** Messages that inform a Confirmable message could not be processed.

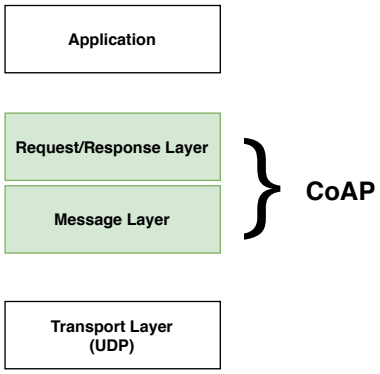


Figure 2.10: CoAP Layers and their positioning on the network stack.

A Confirmable message is reliable and is sent repeatedly at increasing intervals, until the reception of an Acknowledgment (Figure 2.11), a Reset, or if the retransmission counter has reached maximum its maximum value. Non-Confirmable messages are neither acknowledged nor rejected, even if they are not processed (Figure 2.12). Non-Confirmable messages are unreliable and do not contain critical information. Still, they have a unique Identifier (ID).

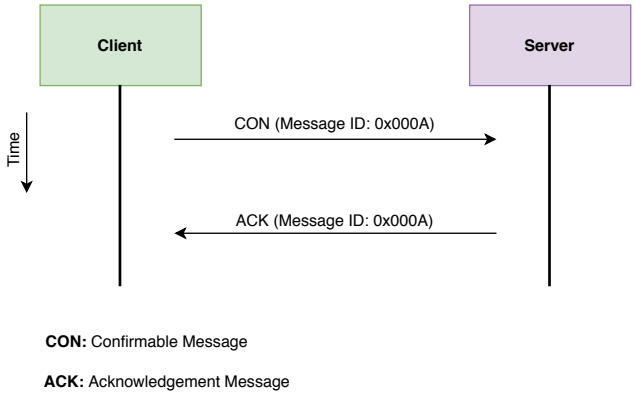


Figure 2.11: CoAP's Confirmable Messages exchange.

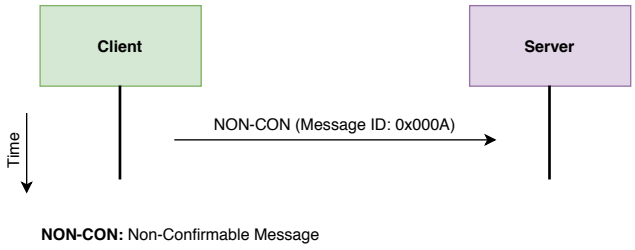


Figure 2.12: CoAP's Non-Confirmable Messages exchange.

The Request/Response layer deals with requests (methods previously mentioned) and response codes to these methods. Requests and responses are linked to one another due to a Token Option. The Token Option is used to match the request and response from the client/server independently from the Message-ID, providing further context.

2.4.2 uIP Network Stack

Initially developed by Adam Dunkels, uIP is intended for small computers. Its goal is the reduction of RAM and ROM usage. Therefore, uIP is smaller than any existing TCP/IP stacks available [60]. However, memory usage is configurable and depends on some factors. This stack is an adaptation of essential elements of the TCP/IP protocol suite, implemented as a small network stack [60]. This network stack is supported by all Contiki's distributions.

uIP buffers its data in RAM memory, allowing easy and quick data gathering, for eventual retransmissions, if needed. It resorts global variables to keep stack's usage low since they are more efficient at execution time and memory usage. Size-wise, uIP is small, both in compiled code size and memory usage resorting to around 300 bytes of memory [60]. As previously referenced, memory usage is configurable and depends on two factors, the maximum number allowed of active connections and the maximum packet size. Three of the underlying protocols of TCP/IP are implemented by uIP. These are IP, TCP, and Internet Control Message Protocol (ICMP). Any other protocols can be implemented if desired; however, they must be implemented as a device driver or an application, in case they are higher-level protocols [60].

2.4.3 OpenWSN Network Stack

Developed in 2010, OpenWSN is an open source implementation of the IEEE 802.15.4e standard and especially suited for constrained devices. It comprises a IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) network stack [11], and it is fully integrated with TSCH, 6LoWPAN, CoAP, and RPL, which provides excellent reliability [61]. The targeted device classes are class 0 and class 1, while also providing some support for class 2 devices. Despite being based on IEEE 802.15.4e, it lacks some procedures defined by IEEE 802.15.e, such as security ones [62]. This network stack has been ported to 32-bit Cortex-M architectures [63] and 11 hardware platforms [64, 65]. The OSes supporting this network stack are OpenOS, FreeRTOS and RIOT [66].

This stack has a higher degree of complexity than the small-coded uIP network stack. Figure 2.13, based on [67] and [68], represents the OpenWSN network stack layers and their positioning. A particular trait of this network stack is a module useful for all layers called Cross-Layers, which crosses all layers from the MAC to the transport layers.

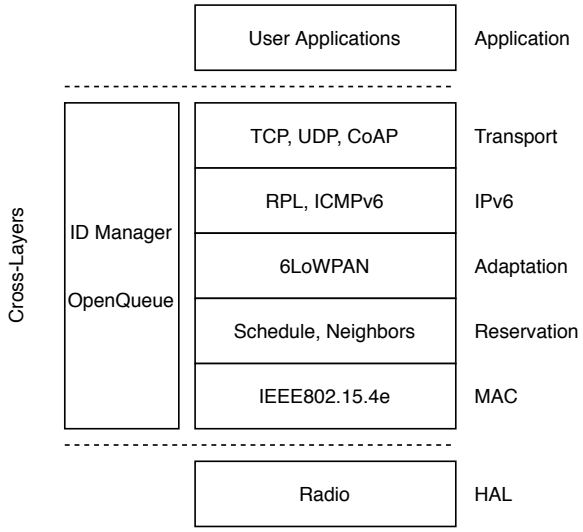


Figure 2.13: OpenWSN network stack layers.

3. Platforms and Tools

The multiple tools used throughout the development of the project and the platforms running the code are described next.

The chapter is organized in the following way: firstly, the several pieces of hardware (boards) are presented. Each hardware is introduced, explained, and corresponded with an illustrative image. In the second half of this chapter, the software used is described, and its details explained.

3.1 Texas Instruments SmartRF06 Evaluation Board Kit

The SmartRF06 Evaluation Board (EB), depicted in Figure 3.1, is part of Texas Instruments' development kits for Low Power RF ARM Cortex®-M based System on Chips (SoCs). According to [69], between many others, this board features an analog Ambient Light Sensor (ALS); four general-purpose Light Emitting Diodes (LEDs); a micro Secure Digital (SD) card slot for connecting external flash devices; an accelerometer; five push-buttons; a 128x64 pixels Liquid Crystal Display (LCD); a current sensing unit for current consumption measurements; options for power supply, such as Universal Serial Bus (USB), batteries or an external power supply; and so on.

The presented board was used to deploy applications with the Contiki OS and the uIP network stack. It was also used exclusively to perform tests on the purposed solution for this thesis.

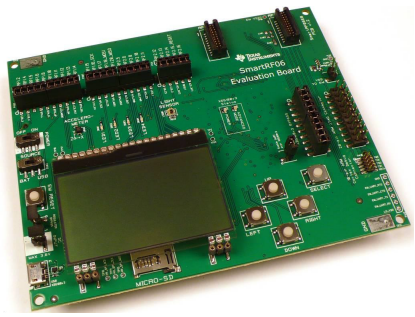
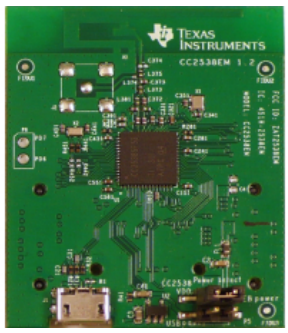


Figure 3.1: Texas Instruments SmartRF06 EB.

3.2 Texas Instruments CC2538 Evaluation Module Kit

Armed with an Advanced Reduced Instruction Set Computer Machine (ARM) Cortex-M3 based Microcontroller Unit (MCU) system, this Evaluation Module (EM), presented in Figure 3.2a, has a 32kB RAM and up to 512kB on-chip flash. The main focal point of this EM is its IEEE 802.15.4 radio. This feature enables network stacks handling as well as Over the Air (OTA) downloads. It also has hardware security accelerators, providing authentication and encryption with no need for CPU processing. This add-on board has multiple low-power modes [70].

In combination with Texas Instruments SmartRF06 EB (Figure 3.1), it enables all previously mentioned features to the EB, which is essential for IoT connectivity. Figure 3.2b presents an image of the SmartRF06 EB and CC2538 EM coupled, with an antenna's addition to increase signal range. This EM was used in conjunction with the SmartRF06 EB. When both are connected, the EM allows network stack handling functionalities to the latter, which was critical to this thesis.



(a) Texas Instruments CC2538 EM.

(b) Texas Instruments SmartRF06 EB and Texas Instruments CC2538 EM coupled.

Figure 3.2: The radio module (Figure 3.2a) and the host controller coupled to the radio module (Figure 3.2b). When connected, the CC2538 EM provides radio functionalities to the host controller.

3.3 Texas Instruments CC2520 Evaluation Module Kit

CC2520 EM, presented in Figure 3.3, is an IEEE 802.15.4 compliant radio with a 250 kilobit per second (kbps) data rate for the 2.4GHz unlicensed Industrial, Scientific, and Medical (ISM) band. It provides a vast set of features that reduce the load on the host controller, such as data buffering and encryption, frame handling, frame timing information, etc.

It is typically used in conjunction with a microcontroller a host. When coupled with such host, it enables IoT connectivity, due to its radio features.

This EM was used in conjunction with the STM32L476G-Discovery. When both are connected, the EM allows network stack handling functionalities to the latter.

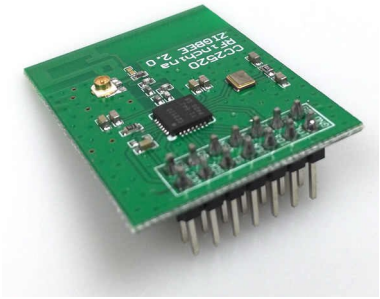


Figure 3.3: Texas Instruments CC2520 Radio EM.

3.4 STMicroelectronics STM32L476G-Discovery

Being part of the famous STM32L4 family, this board, STM32L476G-Discovery, depicted in Figure 3.4, possess ultra-low-power capabilities. To complement this feature, it has numerous peripherals with low-power consumption, such as Universal Asynchronous Receiver-Transmitters (UARTs), timers, etc. and analog peripherals like Operational Amplifiers (op-amps), comparators, LCDs, Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs) [71]. It features a 1Megabyte (MB) flash memory and 128kB RAM memory. Its LCD has 24 segments; it has seven LEDs; a push-button; a four-direction joystick; stereo with jack output; an accelerometer; a magnetometer; a gyroscope; a microphone; a power-consumption sensor; etc.

The presented board was used to deploy applications with the Contiki OS and the OpenWSN network stack.

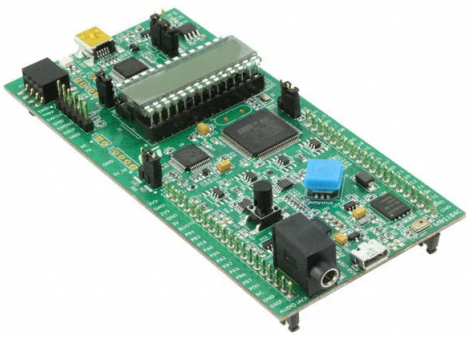


Figure 3.4: STM32L476G-Discovery board.

3.5 Thread-Metric Benchmark Suite

The Thread-Metric Benchmark Suite [72] is a free, open-source, benchmark suite intended for Real-time Operating System (RTOS) performance measurements [73]. Its working principle is based on tests performed on the target system, examining different aspects of it. According to Silva *et al.*, eight tests are performed; these are:

Basic Processing This test consists on one single thread executing mathematical operations, consecutively, for a period of time. Afterward, the result is the number of times this action was performed.

Cooperative Context Switching This test consists on the concurrent execution of five threads, all with the same priority, and every single thread records the number of times they execute. Afterward, the result is the sum between all five counts.

Preemptive Context Switching This test consists on the sequential execution of five threads, all with different priorities, and each thread resumes the following one with a higher priority. Every thread records the number of times they execute. Afterward, the result is the sum between all five counts.

Interrupt Processing This test consists on one single thread running, then being interrupted and resuming after the interruption. Afterward, the result the number of times the Interrupt Service Routine (ISR) was reached, and the thread was executed.

Message Passing This test consists on one single thread transmitting a message to the messaging queue and retrieving it itself. Afterward, the result is the number of send/receive cycles the thread went through.

Semaphore Processing This test consists on one single thread *getting* and *releasing* a semaphore for a period of a predetermined time. Afterward, the result is the number of times this action was performed.

Memory Allocation and Deallocation This test consists on one single thread allocating (acquiring) and deallocating (releasing) memory blocks. Afterward, the result is the number of times this action was performed.

3.6 Contiki-NG Operating System

The open-source, cross-platform Contiki-NG OS has a low code footprint and low memory usage that suits the resource-constrained devices in the IoT. Being an open-source solution makes it ideal for the current thesis since it does not involve any associated monetary costs. Also, since it is oriented for low-end IoT devices, it fully fits the requirements imposed by the chosen hardware boards and modules. It is thoroughly documented, minimizing errors stalls resulting in faster code production, and, on top of this, the Contiki-NG is an extensively used OS, thus it is a state of the art technology, making it more relevant for this thesis.

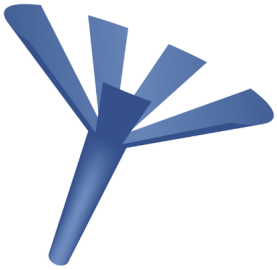


Figure 3.5: Contiki-NG logo.

4. Design

This Chapter presents the design of the solution implemented in this work. It lays all fundamentals required to fully understand what was done in order to make this project come to life. It is divided into three main Sections.

The first Section focuses on the challenges faced when designing the solution. It pinpoints each main obstacle that was needed to overcome to produce the final result. It presents a theoretical explanation of every one of them and why they were relevant. Finally, at the end of the Section, the solution is succinctly explained and how it overcomes the obstacles. The second Section is a purely theoretical Section that focuses on agnosticism. It thoroughly explains the idea of agnosticism, which is crucial to understand the following Section. The third and final Section targets, solely, the solution from a design point of view. It comprehensively presents it with great detail on how it was planned, where it positions itself in the system, and the mechanisms to make it work.

4.1 Design Goals

The goal of this project is to integrate multiple network stacks with a specific OS. However, a network stack has a significant impact on the whole system. From its power consumption to its reliability, supported protocols, etc., the features of a network stack can be plenty. It may be a light network stack, a more complex one, or even one with a wide range of different protocols available for each of its layers. Besides that, some network stacks are more suited for some purposes than others (WSNs, for example). Similarly, the OSes also have characteristics that highly differentiate between each other and make them suited for different scenarios. So, the job of merging multiple and different network stacks with an OS is not such a trivial and straightforward task. This task can be divided into several goals, these are:

- The OS and network stacks must be oriented for low-end IoT devices.
- The combination of the OS and the network stacks must comply with low-end IoT devices resource constrictions.
- The network stacks must be independent from any OS.
- Both network stacks can not perform simultaneously.

One focal point is the orientation for low-end IoT devices. This restriction to constrained devices reduces the number of embedded OSes available to perform on these types of devices. Considering that the OS plus network stacks operate on a low-end device, it can not require high amounts of memory, because low-end devices are very scarce in resources by their nature.

Since the aim is to integrate multiple network stacks with a single OS, the network stacks added to the project must be external ones. The uIP network stack is an open-source, external, extremely lightweight, and widely researched network stack. Along with the uIP network stack, another network stack highly used is OpenWSN. It is open-source, well documented, suited for low-end devices of classes 0, 1, and 2, and even the notorious RIOT OS uses it. Therefore, the uIP network stack and OpenWSN network stack were chosen to integrate the project.

One of the most known, widespread, open-source, and used OS, which carries the uIP network stack, is Contiki OS. Therefore, the OS's choice is Contiki.

A significant obstacle to implementing a solution for integrating multiple network stacks to a single OS is that both network stacks can not process the same data set at the same time. This is because if they happen to perform and process the same physical interface simultaneously, besides an enormous power waste due to the processing both network stacks require, collisions are extremely likely to occur. A collision of trying to access the same resource simultaneously or, in an even worst scenario, they access the same resource concurrently, leading to incorrect results, culminating in data corruption. Since these network stacks were added as a module, they must also be removable as a module, and being intrinsically connected to the OS hinders this task.

The solution for this task is the creation of an Agnostic Layer. It is a layer that is positioned between the OS and both the network stacks and manages interactions between them. It manages which network stack should be performing at which given point, enabling it and disabling the other

one. It also removes any ties with the OS by the network stacks, since both network stacks are not directly connected to the OS and are, on the other hand, connected to the Agnostic Layer. The Agnostic Layer is a lightweight layer developed in software. Since everything comes at a cost, this advantage of abstracting through agnosticism the OS from the network stacks results in heavier and slower program execution, with more clock cycles required and with performance degradation.

4.2 Agnosticism

The agnosticism presented throughout this dissertation revolves around abstracting the system of specific elements. This means that the system, at the point of agnosticism, does not care what protocols were/are used to transmit or receive the data, the interface used for the data to flow, the programming paradigm used, the device(s) or chip(s) active, the power consumption, etc. The system behaves like there is a black box, with no information available to it. What happens inside the black box is not of the system's concern. Its only concern is the input/output connections if it has to communicate with it.

Figure 4.1 represents how a system views its environment. System 1 is aware of how its sensors (Sensor 1, Sensor 2, and Sensor 3) work and are currently performing. The same goes for its database, memory, and Arithmetic Logic Unit (ALU). However, for both the external sensor and System 2 connected, it does not care about their intricacies; it just cares about what concerns it, specifically its connections to these systems (Inter-Integrated Circuit (I^2C) and Serial Peripheral Interface (SPI)).

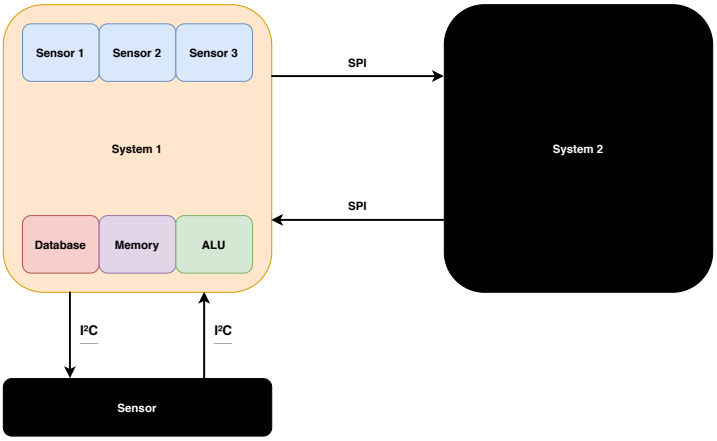


Figure 4.1: An environment of different systems. The figure represents the view by System 1.

A similar panorama goes for System 2, as depicted in Figure 4.2. To System 2, System 1 is a black box. It does not know about System 1's internals, connections, or working principles. Its only concern about System 1 is the communications that they establish through an SPI communication protocol. Moreover, it is worth noting that as System 1 is not aware of System 2's internals or connections (Keyboard), the same goes for System 2 as it is not aware of System 1's connections (external sensor).

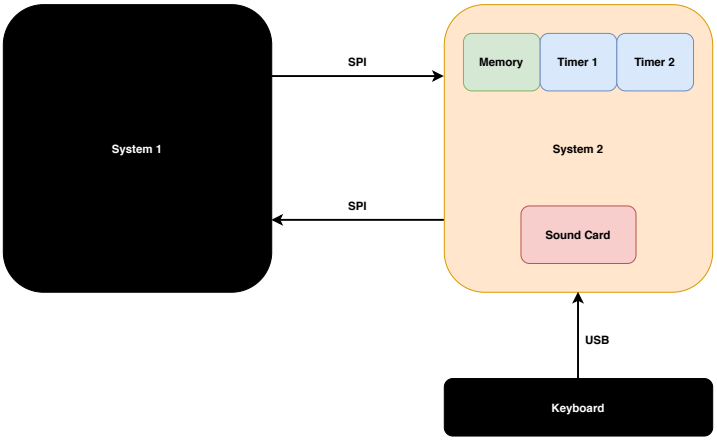


Figure 4.2: An environment of different systems. The figure represents the view by System 2.

The environment on which these two systems are inserted is shown in Figure 4.3. It represents the total scenario of the environment, not viewed by any system, with no agnosticism perspective. These small examples show the idea of agnosticism as one system does not know or care about the functioning of the other.

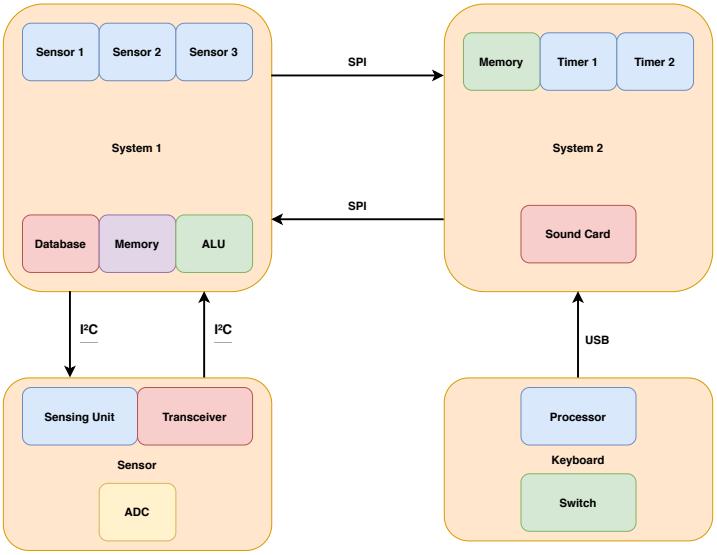


Figure 4.3: An environment of different systems.

4.3 Agnostic Layer

This thesis aims to mitigate the dependency between the OS and the deployed network stack. This is done through a set of agnostic APIs that facilitate the integration of a given OS with one or more network stacks. The set of agnostic APIs creates a layer-like format by positioning themselves between the OS and the network stack. Similarly to the layers of a network stack, these APIs are organized by individual layers that stack on top of each other, creating a layer-set composed of single layers. This is called the Agnostic Layer. From this point on to the end of the current chapter, an in-depth view of the Agnostic Layer is presented and explained, as well as the hows and the whys of connecting several network stacks to a single OS.

This segment is divided into three phases. The first explains where this set of agnostic APIs operates on. It is explained why this approach was used and how it can enable several network stacks to operate with a single OS. The second part revolves around the functions that allow this Agnostic Layer to perform. The last part gives an overview of the network stacks used to develop and test this project and why they were used.

4.3.1 Positioning in the System

Typically, the OS is intrinsically connected to the network stack, as depicted in Figure 4.4. The OS is fully aware of it. However, this approach hinders the task of adding more external network

stacks to the OS.

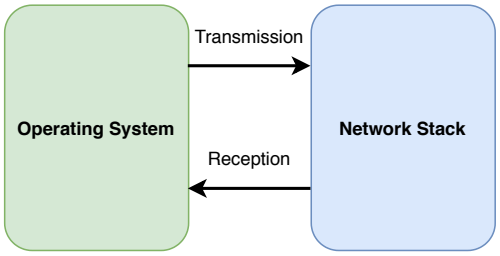


Figure 4.4: Diagram of communications between the OS and network stack.

The chosen approach was to introduce an agnostic layer, positioned between the OS and the network stack, as seen in Figure 4.5. This removes the OS out of the scenario of establishing communications with the network stack, which is the Agnostic Layer's job. It creates agnosticism for the OS, only managing data packets in specific formats. The job of managing the format of said data packets falls on the Agnostic Layer, which interacts directly with the OS and the network stack.

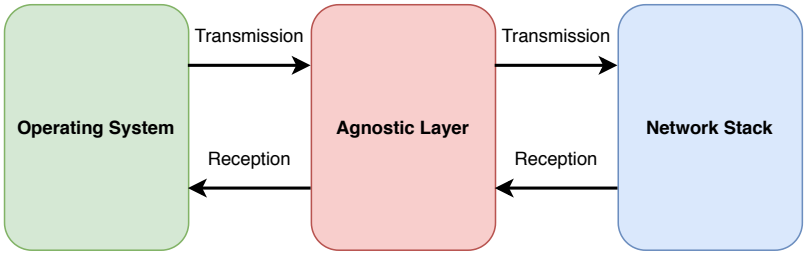


Figure 4.5: Diagram of communications between the OS, Agnostic Layer and network stack.

When implemented, more external network stacks can be deployed, since the OS is not intrinsically connected nor aware of them. Figure 4.6 reveals this idea showing the network stacks used in the project (uIP network stack and OpenWSN network stack).

The OS is not aware of both the stacks. It just handles data packets in the format it is programmed. The Agnostic Layer is responsible for communicating with the network stacks and managing the formats the packets assume accordingly.

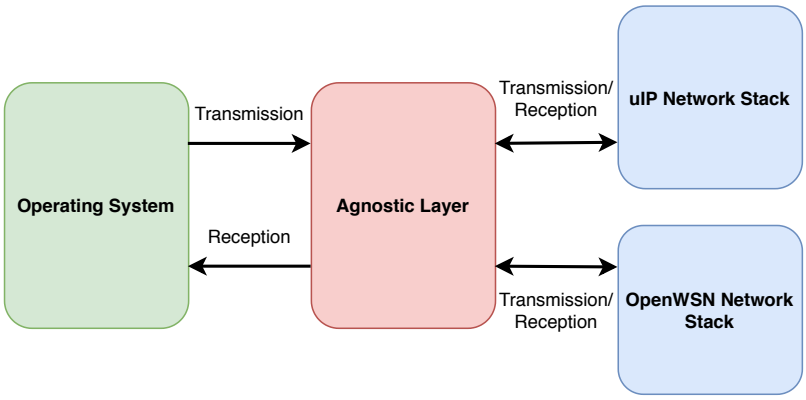


Figure 4.6: Diagram of communications between the OS, Agnostic Layer and two network stacks (uIP and OpenWSN).

The two network stacks must be selected before runtime. A mechanism was designed to perform this function. It enables one of the two available network stacks and disables the other. All communications flow through the enabled network stack. The network stack selection is represented in Figure 4.7 and Figure 4.8. As portrayed in the figure, the network stack represented in blue is currently active within the system, while the gray one is the network stack disabled.

To the OS, no change has been made as it is only connected to the Agnostic Layer.

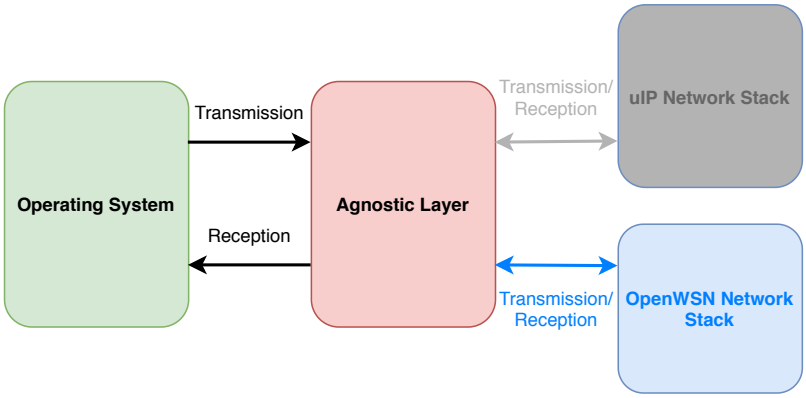


Figure 4.7: OpenWSN network stack enabled.

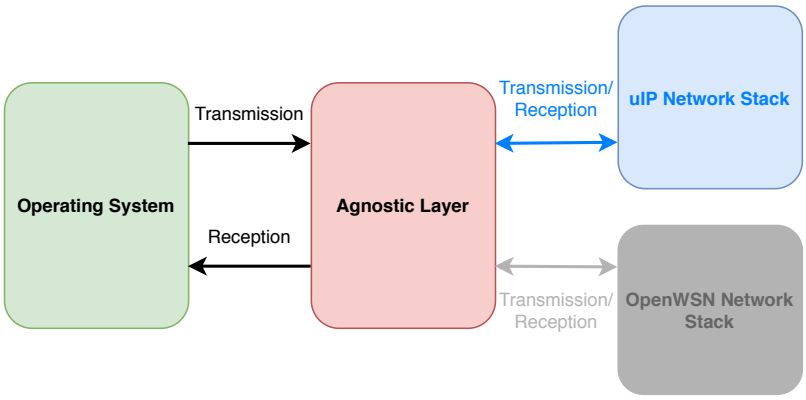


Figure 4.8: uIP network stack enabled.

4.3.2 Remapping Functions

In order to switch between network stacks, enabling one stack and disabling the other, remapping functions are necessary. Present in the Agnostic Layer, these functions redirect the program flow to the selected network stack.

The remapping functions of a particular network stack layer are organized by a structure that aggregates them, creating Agnostic Layer layers, as represented in Figure 4.9, each one corresponding to a layer of the network stack. There are several remapping functions for each layer of the Agnostic Layer.

With this design, the Agnostic Layer mimics a generic network stack by emulating the behavior of one. Since the OS is unbeknownst of this and performs as if the Agnostic Layer is a generic network stack, the reception and transmission of data packets to/from the OS remain unchanged, thus creating agnosticism within the system.

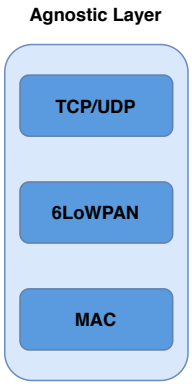


Figure 4.9: Exemple of different layers forming an Agnostic Layer.

Remapping functions are invoked at the input/output of the layers of the network stack, as presented in Figure 4.10. The figure shows a possible network stack, and, in the said network stack, the dots in red are where the remapping functions are invoked. As mentioned before, these points are the inputs and outputs of each single network stack layer.

This format implies that before and after executing through a layer (for transmission or reception), the program is redirected to the Agnostic Layer. This allows two things: Each layer becomes independent from one another, and program flow can be traced more easily.

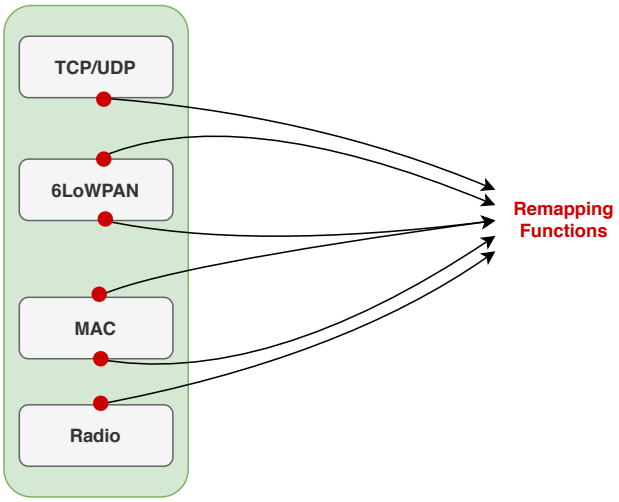


Figure 4.10: Remapping functions positioning in the network stack.

With this approach, the Agnostic Layer guarantees that the network stack layer runs through entirely unless an interruption occurs. A feature that is enabled with this method is that a layer from the network stack can be switched with another with no significant repercussions or heavy coding by the developer, since they are, to some degree, independent from the other layers that compose the same network stack. Considering this approach, a hybrid network stack can also be created by merging some layers from one network stack and other complementary ones from another network stack(s).

4.3.3 uIP Network Stack and OpenWSN Network Stack

To implement the Agnostic Layer two network stacks were integrated: the uIP and the OpenWSN network stacks.

The choice of the network stacks was simple: the uIP network stack is an external network stack with no significant dependencies with an OS. It is an open-source, well-known, well-documented, and researched light network stack. However, it can be found connected with Contiki, by default, in Contiki's distribution. So the goal with this network stack is firstly to remove all dependencies between Contiki and the network stack itself. The next step is to identify the merging points with the OS since not all layers are connected to it. Some layers call subsequent layers in sequential order, not needing to pass the program flow to the OS. The last step is to insert Agnostic Layer APIs in those merging points, having full access to both the Contiki OS and uIP network stack, and becoming the bridge that connects these two systems. OpenWSN network stack, on the other hand, is not connected to Contiki OS by any means, so the obstacle with this network stack is integrating it into the existing project. A first step revolves in connecting the external network stack directly to the Contiki OS. Similarly to the uIP network stack, merging points with the OS must be separated to insert the Agnostic Layer in between, making the Agnostic Layer the link that connects both the uIP network stack and the OpenWSN network stack with the Contiki OS.

5. Implementation

The current Chapter focuses on the project's implementation. It mainly centers on the practical aspects of the problems at hand and the solutions to them. This Chapter is divided into three sections: Network Stack Selection, Network Stack Configuration, and Agnostic Layer. These are three fundamental topics to implement the abstraction of the OS to the network stacks and vice-versa through the Agnostic Layer.

The first Section regards the network stack selection, a design challenge stated in the previous Chapter. It presents the solution to it, how it was implemented, and implications to the Agnostic Layer. The second Section, Network Stack Configuration, targets some configurations needed in order to integrate the Agnostic Layer with Contiki OS. The third and final Section, Agnostic Layer, spotlights important functions to the system.

5.1 Network Stack Selection

As the focus of this thesis is the integration of multiple network stacks with an OS, and since two network stacks can not co-operate (if so, collisions would occur), a mechanism that enables the selection of a network stack must be present and available to the user. This feature must guarantee that only one network stack is processing the data from a single hardware interface.

The selection of the network stack is made through a set of macros. As exhibited in Listing 5.1, the user can select the desired network stack by either commenting or uncommenting line 8, as instructed by code comments, thus enabling the uIP network stack or the OpenWSN network stack, respectively. If more network stacks were to be added to the project, this would imply minor alterations to the code in order for this mechanism to continue to work. This ensures a simple, practical, and user-friendly way to commute between network stacks. Other approaches could be resorted to in order to perform this network stack choice. However, the mechanism hereby

described allows a user-friendly approach to the network stack selection since it does not require a deep understanding of the Agnostic Layer's functioning to be able to select the network stack desired. Besides, this solution presents itself as a low overhead solution since the total cost to the system is the definition of a single macro.

```
1 /*----------------- Network Stack Selection ---------------------*/
2 /**
3 * \note To select desired network stack:
4 *
5 *   uIP Stack: Uncomment "#define uIP_Stack"
6 *   OpenWSN Stack: Comment "#define uIP_Stack"
7 */
8 #define uIP_Stack
9
10 #ifdef uIP_Stack
11 #undef OpenWSN_Stack
12 #endif
13
14 #ifndef uIP_Stack
15 #define OpenWSN_Stack
16 #endif
17 /*---------------------------------------------------------------*/
```

Listing 5.1: Network stack selection method.

When a network stack is chosen through the method above, the disruption imposed on the system affects only the Agnostic Layer, without making the OS aware of this modification. This change of network stack disables one network stack and enables the other one. The OS performs normally, unbeknownst of any difference to the system, since it views the Agnostic Layer as his network stack. However, the Agnostic Layer is aware of the selected network stack and will forward and receive the data packets to/from it, respectively. From that point on, all receptions and transmissions flow through the Agnostic Layer and the newly selected network stack. The data packets arrive (at reception time) and are sent (at transmission time) to/from the OS unchanged,

no matter which network stack was chosen.

5.2 Network Stack Configuration

In the way Contiki is built, the network stack is customized by accessing a group of configuration files. These files allow the configuration of the diverse protocols of each layer in the network stack. Since there is more than one protocol available for some layers, the Contiki OS allows this management of different protocols using that group of configuration files.

In order to allow the Agnostic Layer to connect with the Contiki OS, some configurations had to be made. The Agnostic Layer is composed of several layers, mimicking the structure of a regular network stack. So, the method to integrate the Agnostic Layer with the Contiki OS is by making each layer of the Agnostic Layer correspond to a layer of Contiki's network stack. Instead of invoking a determined layer of a regular network stack, the Contiki OS calls the corresponding layer of the Agnostic Layer instead. This is achieved by accessing the configuration files before compilation and enabling each layer of the Agnostic Layer as the protocols selected to perform. With this method, when the Contiki OS needs to invoke a particular protocol from a network stack's layer, it calls the equivalent layer but from the Agnostic Layer instead. Analyzing at a larger scope, all the network stack's layers are replaced by Agnostic Layer's layers. This allows the Agnostic Layer to become the Contiki OS's very own network stack.

A snippet of one configuration file in which the MAC and 6LoWPAN layers, for the Agnostic Layer, are configured to integrate the Contiki OS is presented in Listing 5.2. In this example, it is worth noting the commented lines 3 and 11, that previously integrated uIP's layers to the project. However, as presented in lines 4 and 12 the configuration was set to include layers MAC and 6LoWPAN of the Agnostic Layer, respectively.

```
1 #ifndef NETSTACK_CONF_NETWORK
2 #if NETSTACK_CONF_WITH_IPV6
3 //#define NETSTACK_CONF_NETWORK sicslowpan_driver
4 #define NETSTACK_CONF_NETWORK agnostic_sicslowpan_driver
5 #else
6 #define NETSTACK_CONF_NETWORK rime_driver
```

```
7 #endif /* NETSTACK_CONF_WITH_IPV6 */
8 #endif /* NETSTACK_CONF_NETWORK */
9
10 #ifndef NETSTACK_CONF_MAC
11 //#define NETSTACK_CONF_MAC      csma_driver
12 #define NETSTACK_CONF_MAC     agnostic_csma_driver
13 #endif
```

Listing 5.2: Configuration of MAC and 6LoWPAN layers for the Agnostic Layer.

Another example, among many, of more configurations that must be made in order for the system to view the Agnostic Layer as the network stack, is the integration of the data structures that compose a layer of the Agnostic Layer. In order to add each data structure to the project, they have to be added individually, for the system to accept these new data structures as layers. Listing 5.3 shows this, as previous data structures corresponding to a former network stack have been disabled and Agnostic Layer taking their part.

```
1 //extern const struct network_driver NETSTACK_NETWORK;
2 extern const struct agnostic_network_driver NETSTACK_NETWORK
3 //extern const struct mac_driver NETSTACK_MAC;
4 extern const struct agnostic_mac_driver NETSTACK_MAC;
```

Listing 5.3: Data structures that compose the MAC layer and the 6LoWPAN layer of the Agnostic Layer taking their part as the new network stack data structures.

5.3 Agnostic Layer

From this point on, until the end of the chapter, the focus is on relevant functions of the Agnostic Layer. There are two main groups. The first comprises the data structures that compose each layer of the Agnostic Layer. These data structures assemble a group of remapping functions that are essential for the Agnostic Layer. The second group is the remapping functions, as

previously introduced. These functions are the ones responsible for the execution of the network stack configured by the user.

5.3.1 Agnostic Layer Data Structures

As previously mentioned, the remapping functions are aggregated together by a data structure. This data structure is composed of all the remapping functions of a particular Agnostic Layer layer. An example of such data structure is shown in Listing 5.4. This example presents the structure that integrates the remapping functions for the MAC layer, and how this layer is set up. Since the Agnostic Layer's fundamental concept revolves around agnosticism, the main functions of a given layer (functions common to several network stacks and essential to the performance of the layer) are the ones that integrate the remapping functions. The network stack's specific functions do not make sense of being part of the Agnostic Layer since their implementation is oriented to their network stack, which counteracts the concept of agnosticism. Thus, the network stack's specific functions are located at their correspondent network stack implementation, not in the Agnostic Layer.

```
1 /**
2 * The structure of the MAC layer for the Agnostic Layer.
3 */
4 struct agnostic_mac_driver {
5 char *name;
6
7 /** Initialize the MAC driver */
8 void (* init)(void);
9
10 /** Send a packet from the packetbuf  */
11 void (* send)(mac_callback_t sent_callback, void *ptr);
12
13 /** Callback for getting notified of incoming packet. */
14 void (* input)(void);
15
16 /** Turn the MAC layer on. */
```

```
17 int (* on)(void);
18
19 /** Turn the MAC layer off. */
20 int (* off)(void);
21 };
22
23
24 /*---------------------------------------------------------------------*/
25 /**---- The agnostic NETSTACK data structure for MAC layer ----**/
26 extern const struct agnostic_mac_driver agnostic_csma_driver;
27 /*---------------------------------------------------------------------*/
```

Listing 5.4: Structure that assembles the remapping functions of the MAC layer of the Agnostic Layer.

5.3.2 Remapping Functions

As previously mentioned, the remapping functions are crucial to the Agnostic Layer, since these functions are responsible for redirecting the program flow to the selected network stack. Remapping functions are typically called at the input/output points of the layers of the network stack.

Listing 5.5 exhibits the idea presented before. At the transmission of a individual packet, the 6LoWPAN layer passes the program execution, in the function *send_packet*, to the subsequent layer. At this point, a Remapping Function takes his place and redirects the program flow to the Agnostic Layer, allowing it to take full control of the processing of the packet of the network stack. In this example, the Remapping Function is *NETSTACK_MAC.send* and redirects the program flow to the function *send* of the MAC layer of the Agnostic Layer.

```
1 static void send_packet(linkaddr_t *dest)
2 {
3   packetbuf_set_addr(PACKETBUF_ADDR_RECEIVER, dest);
4
5 #if NETSTACK_CONF_BRIDGE_MODE
```

```
6   packetbuf_set_addr(PACKETBUF_ADDR_SENDER,(void*)&uip_lladdr);
7 #endif
8
9   /* Remapping Function */
10  NETSTACK_MAC.send(&packet_sent, NULL);
11
12  watchdog_periodic();
13 }
```

Listing 5.5: Remapping Function called at the output of the 6LoWPAN layer of a network stack.

Listing 5.6 displays the body of a Remapping Function of the MAC layer of the Agnostic Layer, more concretely the initialization function. Two key points of the presented listing: how remapping functions of the Agnostic Layer are called up and how to redirect the program flow through the selected network stack.

At line 10 of the listing previously mentioned, there is a call for a Remapping Function of the Agnostic Layer. It happens that the Remapping Function called is located at the same layer as the calling function, namely at the MAC layer of the Agnostic Layer. To present a different example, Listing 5.7 displays a calling inside the Agnostic Layer from remapping function *Send(mac_callback_t sent_callback, void *ptr)* of the MAC layer to either appropriate function of the selected network stack. According to the selected network stack, the program flow runs through either the uIP MAC's own send function or to OpenWSN's one. Besides these two examples, callings to remapping functions from other locations, such as network stacks or the OS, occur as well.

```
1 static void Agnostic_Init(void)
2 {
3 #ifdef uIP_Stack
4   csma_output_init();
5   NETSTACK_MAC.on();
6 #elif defined OpenWSN_Stack
7   ieee154e_init();
```

```
8 #endif
9 }
```

Listing 5.6: Remapping Initialization Function of the MAC layer of the Agnostic Layer.

```
1 static void Agnostic_Send(mac_callback_t sent_callback, void *ptr)
2 {
3 #ifdef uIP_Stack
4   csma_output_packet(sent_callback, ptr);
5 #elif defined OpenWSN_Stack
6   activity_ti1ORri1();
7 #endif
8 }
```

Listing 5.7: Remapping Send Function of the MAC layer of the Agnostic Layer.

Inside the Agnostic Layer's remapping functions, there are macros that agglomerate code for each of the network stacks added. In Listing 5.6, Listing 5.7, and Listing 5.8, it is possible to identify two sections, one for each network stack added, that gathers code, which may or may not be equal from one another, for the behavior pretended for each network stack. This is, in the listings previously mentioned, from the preprocessor directives *#ifdef uIP_Stack* to *#elif defined OpenWSN_Stack*, the code in between regards the code for the functioning of the uIP network stack. Similarly, from the preprocessor directives *#elif defined OpenWSN_Stack* to *#endif*, regards the code for OpenWSN network stack. The coding for each network stack may or may not be equal from one another as they differ in their implementation. This is the purpose of the referred macros, to enable the execution for the network stack previously chosen, by modifying the code presented in Listing 5.1. What happens is the selection of the network stack to execute, not allowing both to execute at the same time, thus preventing collisions of the two network stacks.

```
1 static unsigned short Agnostic_Channel_Check_Interval(void)
```

```
2 {
3 #ifdef uIP_Stack
4   if(NETSTACK_RDC.channel_check_interval) {
5     return NETSTACK_RDC.channel_check_interval();
6   }
7   return 0;
8 #elif defined OpenWSN_Stack
9   OpenWSN_Channel_Check_Interval();
10 #endif
11 }
```

Listing 5.8: Remapping Function Agnostic_Channel Check_Channel_Channel Interval of the MAC layer of the Agnostic Layer.

6. Evaluation and Results

The current Chapter concerns the evaluation of the implemented software-based Agnostic Layer. Three different types of evaluations were used to analyze the impact of the Agnostic Layer on the system: microbenchmark evaluation tests, memory footprint and Thread-Metric evaluation tests. The first evaluation test enables gathering information about the overhead added to network stack functions caused by the Agnostic layer. The second evaluation test aimed to gather information about the memory overhead imposed by the integration of the Agnostic Layer. The third and final evaluation test evaluates the Agnostic Layer's impact on the system's performance.

The values gathered by the three tests previously mentioned are both compared to the OS' baseline values to have a comparative point and better understand the influence the Agnostic Layer has on the system when it is integrated.

6.1 Microbrenchmarks on Main Functions

The microbenchmarking process concerns the usage of a single hardware timer to the system. This timer aims to measure the number of clock cycles it takes since the point of invocation of a network stack function until it reaches the core of the function. Performing this evaluation in an unmodified system results in the system's baseline behavior, i.e., the system's standard performance when running normally. With the integration of the Agnostic Layer, the microbenchmarks are expected to present different results compared to their baseline values. This expectation corresponds to the Agnostic Layer's placement between the OS and the network stacks, which vastly increases the amount of instructions the CPU has to process, taking longer clock cycles to reach the function's body.

The tests were performed on two network stack layers, namely, on the MAC and the 6LoWPAN layers with and without the addition of the Agnostic Layer. Additionally, the tests were also performed with and without the packets being transmitted over the network. When packets were being processed, the packet reception rate was around 125 packets per second. In total, eight different types of tests were performed, they were: with packets being transmitted, with and without the inclusion of the Agnostic Layer, and without packets being transmitted, with and without the Agnostic Layer. Each different type of test was executed 1000 times.

Table 6.1 presents the average results gathered from running the microbenchmarks tests. It reunites the results from the tests with/without packets and with/without the Agnostic Layer, both for the MAC and 6LoWPAN layers.

Table 6.1: Average results obtained from performing the microbenchmarks tests.

		6LoWPAN (clock cycles)	**MAC (clock cycles)**
No Packets	No Agnostic Layer	15	15
	With Agnostic Layer	19	19
With Packets	No Agnostic Layer	15	15
	With Agnostic Layer	21	21

Figure 6.1 graphically illustrates the previously displayed results of the microbenchmarks tests. The blue and gray bars represent the tests performed on functions without the Agnostic Layer, whereas the orange and yellow bars represent the same tests but with the inclusion of the Agnostic Layer.

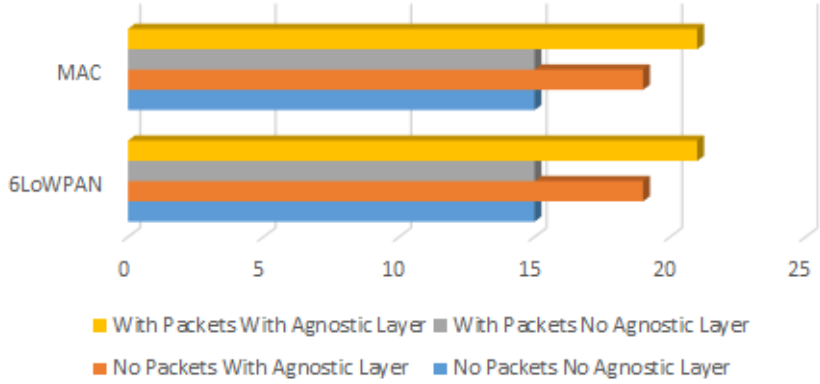


Figure 6.1: Microbenchmarks tests results.

With the integration of the Agnostic Layer, the number of clock cycles taken from the point of invocation of the previously mentioned functions until reaching the function's body is higher than without the integration of the Agnostic Layer, as initially expected. For both layers (MAC and 6LoWPAN), without packets being transferred over the network, and without the integration of the Agnostic Layer, on average, the system took 15 clock cycles to reach the function's body. Immediately after integrating the Agnostic Layer with the system, this number rose to 19 clock cycles. More specifically, the introduction of the Agnostic Layer translates into 26,7% more clock cycles to reach the function's body when compared to the system without it. However, this value may be misleading due to the fact that the Agnostic Layer only increased the time from the point of invocation of the functions until the reaching of the function's body by two clock cycles. This phenomenon is due to the introduction of new instructions in the source code. The additional set of instructions, caused by the inclusion of the Agnostic Layer, require additional time to process, culminating in extra clock cycles necessitated to reach the network stack's function.

If the system is being saturated with the processing of 125 packets per second flowing through the network, while the number of clock cycles required to reach the function's body remains in 15 clock cycles without the Agnostic Layer, the integration of the latter bumps the number of clock cycles required to 21 clock cycles. This increase is equivalent to 40% more clock cycles required to perform the same operation. Again, the value may be deceitful since the increase is in the order of 6 more clock cycles required. This result is due to the fact that when packets are flowing through the network, the radio originates several interruptions. When processing a data packet, if the program flow is somewhere between the Agnostic Layer and the network stack currently active, an interruption redirects the program flow from the processing of said data packet to the

interruption, to be able to physically receive or transmit bytes of data. This interruption halts the system, therefore increasing the number of clock cycles required to process a single data packet.

6.2 Memory Footprint

The memory footprint results aim to concretely provide information about the memory overhead induced by the Agnostic Layer when integrating the latter within the system. Table 6.2 presents the results gathered from analyzing the linker *.map* files generated by the Integrated Development Environment (IDE) after the application's compilation. This table reunites the results gathered (in bytes), both with and without the inclusion of the software-based Agnostic Layer. Regarding the results, the expectation revolved around the integration of the Agnostic Layer increasing the memory footprint of the system in comparison to the system without it. This expectation was substantiated on the basis of the additional instructions necessary to process the Agnostic Layer. As expected, the integration of the Agnostic Layer resulted in flash memory and RAM overhead, as presented in the previously mentioned table and graphically depicted in Figure 6.2. The increase in overhead was in the order of an additional 15 298 bytes of flash memory and an additional 2 427 bytes of RAM required to process the Agnostic Layer.

Table 6.2: Flash and RAM usages.

	Flash Memory (bytes)	**RAM Memory (bytes)**
With Agnostic Layer	67 236	14 988
Without Agnostic Layer	51 938	12 561

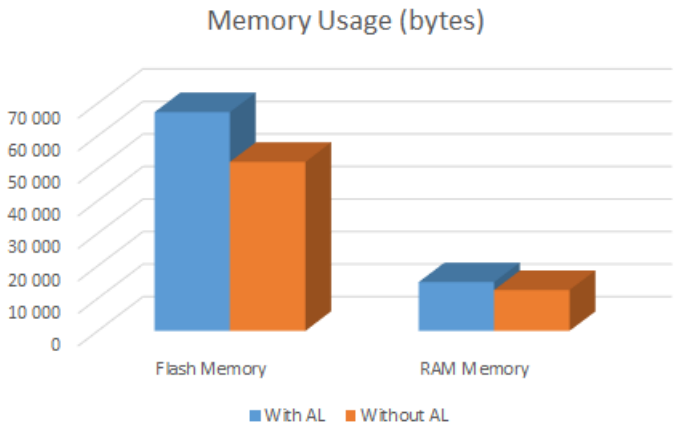


Figure 6.2: Flash and RAM usages with/without the Agnostic Layer.

6.3 Thread-Metric Evaluation

The Thread-Metric Benchmark Suite is an open-source, vendor-neutral, free benchmark suite that measures RTOS performance. The goal of using this benchmark suite is to perform a set of tests intended to evaluate how the software-based Agnostic Layer impacts the normal OS performance. Each test produces a score based on the system's performance. Evaluating the system without the integration of the Agnostic Layer results in the baseline behavior of the system. It is expected performance degradation when there is data exchange in the network. Also, some performance degradation is expected when integrating the Agnostic Layer with the system due to the Agnostic Layer's additional processing requirements, which affects the system's performance.

The tests were performed on four different scenarios. Similarly to the microbenchmarks, the tests were performed with and without the addition of the Agnostic Layer as well as with and without the packets flowing through the network. When the data packets were being processed, the packet rate was 125 packets per second.

Table 6.3 presents the results gathered from running the Thread-Metric Benchmark Suite tests. It reunites the results from the multiple tests performed with and without packets exchange, in the network, and with and without the integration of the Agnostic Layer.

Table 6.3: Results obtained from performing the Thread-Metric Benchmark Suite tests.

		Basic Processing	Cooperative Scheduling	Preemptive Scheduling	Message Processing	Synchronization Processing	Memory Allocation
No Packets	No Agnostic Layer	21819,2	1538982,4	1585463,4	1416913,2	1412669,2	1174428,4
	With Agnostic Layer	20843	1404128	1448537	1325486,4	1307280,6	1071731,6
With Packets	No Agnostic Layer	21503	1482059	1522402,8	1365462,4	1356890,8	1117123,6
	With Agnostic Layer	20548,6	1324965,2	1344547,2	1246474,4	1249929,8	1014870,4

Figure 6.3 graphically illustrates the previously displayed results of the Thread-Metric Benchmark Suite. Similarly to the other tests, this test is divided into four different scenarios. On the illustrated graph, the blue and gray bars represent the tests without the Agnostic Layer,

while the orange and yellow represent the same tests but with the integration of the Agnostic Layer to the system.

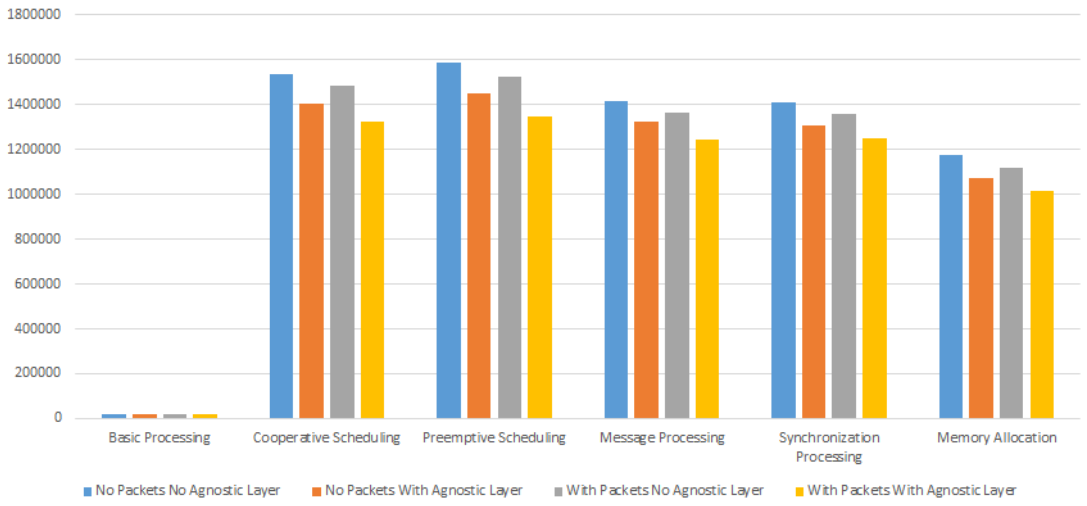


Figure 6.3: Thread-Metric tests results.

According to the Thread-Metric test results, there is a significant performance degradation not only when processing data packets flowing through the network but with the integration of the Agnostic Layer as well. This result dramatically increases when both happen simultaneously.

When compared to its baseline value (without the inclusion of the Agnostic Layer and with no packets flowing through the network), the software-based Agnostic Layer carries an impact of 8,35% on the overall performance of the entire system. This is related to the processing requirements imposed by this layer, as it is a purely software-based layer. Also, as seen in previous tests, the inclusion of the Agnostic Layer increases the total overhead of the system, especially memory-wise. This leads to slower program execution and increased demand of resources, affecting the system's overall performance.

When saturating the system with data packets flowing through the network at a packet rate of 125 packets per second, the performance presented by the system decreases by 4,1%. The decrease in performance is the result of the increased processing by the CPU, required to handle the data packets. Besides, the many interruptions the radio originates, due to the transmission and reception of the data packets, also considerably influence this value. The influence the radio interruptions have on the system revolves around the fact that when a packet has to be physically

received or transmitted by the radio hardware module, the system is halted while processing the ISR. This process takes time, in which the CPU is busy processing the interruption. It is only after the interruption is handled that the system can resume its normal functioning.

The overall degradation of performance of the system skyrockets to 14,2% when on top of the mentioned data packets transmission and reception, is the inclusion of the Agnostic Layer to the system. The Agnostic Layer and the processing of data packets are handled by software, which directly impacts the system's behavior. Besides, and alluding to the microbenchmarking tests, although the inclusion of the Agnostic Layer translates into six more clock cycles to reach the network stack's functions (when data packets are flowing through the network), this number is greatly amplified when consecutively invoking these functions. This phenomenon leads to reduced performance, especially when saturating the system with data packets.

When comparing the results from the Thread-Metric Benchmark Suite tests with and without data packets flowing through the network, and with the inclusion of the Agnostic Layer, there is a difference of 5,85% of the decrease in the performance of the system between the tests. This means that the processing of network data packets shows a more significant decrease in the system's performance when in the presence of the Agnostic Layer. In conclusion, the Agnostic Layer negatively influences in 1,75% the processing of the network data packets. This is the result of the longer clock cycles required to reach the network stack's functions, due to the Agnostic Layer's integration, and the total impact the processing of radio's interruptions presents to the overall performance of the system.

7. Conclusion

Ubiquitous computing and the never-ceasing development and deployment of embedded IoT devices in the network edge brings several challenges. The demand for connectivity and, more importantly, interoperability between devices has led to alternative solutions to enforce these requirements even in low-power, low-cost, low-end IoT devices. A particular level of performance is expected, which is required to process the many events and routines associated with an IoT connection. The connection of IoT devices to the Internet demands the adoption of a network stack. However, since each network stack deploys a custom implementation, typically with varied protocols, the interoperability between IoT devices can become a complex task.

Since a network stack can be added to the OS as a software module, it should be possible to be deployed over any OS. However, the problem arises when integrating a new network stack with the OS. The network stack is usually fully immersed in the OS, creating deep intricacies in between, and, if the OS previously had a network stack connected, it is uprooted to allow other network stacks to take its place. Chosen to accommodate both, it is the OS's duty to juggle between them and to manage the tremendous complexity originated.

The Agnostic Layer is a software-based layer, intended to mitigate the network stack's dependency from the OS and to facilitate the OS's integration with other network stacks without sacrificing its normal behavior while providing full connectivity to the IoT network. The Agnostic Layer's structure is similar to a network stack's structure. It is formed by several layers that stack on top of each other, creating a back to back layered stack. The Agnostic Layer positions itself between the OS and the deployed network stack, allowing to manage the data packets from both the OS and network stack.

Since everything requires balance, so does the integration of said Agnostic Layer. This balance most clearly translates into a trade-off between the Agnostic Layer and the system's performance. With the addition of agnosticism, the system's performance is hindered due to the extra layer of

software added to the code. The trade-off between agnosticism and performance becomes even more noticeable when saturating the system with the processing of data packets. When this happens, the system takes up to 40% more time to reach the network stack's functions when the Agnostic Layer is included. Besides, according to the Thread-Metric Benchmark Suite, the Agnostic Layer has an impact of 8,35% on the overall performance of the entire system.

With this thesis, it was possible to conclude that the Agnostic Layer does indeed add agnosticism to the system, abstracting the OS from the deployed network stacks. The aforementioned layer allows a reduced degree of complexity within the overall system, which eases the addition and removal of external network stacks, which is a desired feature. Lastly, it was possible to conclude that software-based solutions carry a trade-off, typically performance-related wise.

7.1 Future Work

Despite all the work developed and provided throughout this thesis, further upgrades and functionalities can be added to the solution:

- **Support more network stacks:** Since the Agnostic Layer's purpose is to create agnosticism between the OS and the deployed network stack, the inclusion of other network stacks upgrades the Agnostic Layer to a more complete one by being capable of handling a broader range of network stacks.

- **Test each network layer on a different network stack:** Further work would include exchanging one network stack layer by the respective layer of another network stack, and test the system. The goal is to further prove the agnosticism and interoperability by testing the exchange between layers designed for different network stacks.

- **Provide an user-friendly interface system:** Currently, the Agnostic Layer is configurable only by modifying the source code. A user interface, something that currently lacks the Agnostic Layer, would ease the addition and removal of network stacks. On top of this, it eases the management of the software-based layer without requiring a deep understanding of its functioning. Besides the mentioned features, it prepares the product

for future marketing, if intended, as it needs to be user-friendly in order to become a Commercial Off-The-Shelf (COTS) solution.

- **Develop an incompatibility detection mechanism:** Some network stacks are not suited for low-end OSes. Others that may be suited have memory or processing requirements that can not be complied with. For whatever reason it may be, by developing an incompatibility detection mechanism, the network stacks not entirely suited to integrate the Agnostic Layer are discarded. This mechanism would be an add-on to the previously mentioned future feature of developing a user interface.

References

[1] S. Pinto, J. Cabral, and T. Gomes, "We-care: An IoT-based health care system for elderly people," in *2017 IEEE International Conference on Industrial Technology (ICIT)*, pp. 1378–1383, 2017.

[2] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, "Standardized Protocol Stack for the Internet of (Important) Things," *IEEE Communications Surveys Tutorials*, vol. 15, pp. 1389–1406, Third 2013.

[3] R. Porkodi and V. Bhuvaneswari, "The Internet of Things (IoT) Applications and Communication Enabling Technology Standards: An Overview," in *2014 International Conference on Intelligent Computing Applications*, pp. 324–329, March 2014.

[4] T. Gomes, D. Fernandes, M. Ekpanyapong, and J. Cabral, "An IoT-based system for collision detection on guardrails," in *2016 IEEE International Conference on Industrial Technology (ICIT)*, pp. 1926–1931, 2016.

[5] J. Brito, T. Gomes, J. Miranda, L. Monteiro, J. Cabral, J. Mendes, and J. L. Monteiro, "An intelligent home automation control system based on a novel heat pump and Wireless Sensor Networks," in *2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE)*, pp. 1448–1453, 2014.

[6] T. Gomes, N. Brito, J. Mendes, J. Cabral, and A. Tavares, "WECO: A wireless platform for monitoring recycling point spots," in *2012 16th IEEE Mediterranean Electrotechnical Conference*, pp. 468–472, 2012.

[7] R. Khoshnaw[1], D. Doghramachi, and M. Al-Hakeem, "A Review on Internet of Things' Operating Systems, Platforms and Applications," 02 2017.

[8] H. Rajab and T. Cinkelr, "IoT based Smart Cities," in *2018 International Symposium on Networks, Computers and Communications (ISNCC)*, pp. 1–4, June 2018.

[9] Margaret Rouse, Alexander Gillis, Linda Rosencrance, Sharon Shea, and Ivy Wigmore, "Internet of Things." [Online]. Available: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT, Accessed on: Jan. 13, 2020.

[10] T. Harwood, "Internet of Things History." [Online]. Available: https://www.postscapes.com/iot-history/.

[11] Hahm, Oliver and Baccelli, Emmanuel and Petersen, Hauke and Tsiftes, Nicolas, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, pp. 1–1, 12 2015.

[12] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[13] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained node networks." [Online]. Available: https://www.ietf.org/rfc/rfc7228.txt, Accessed on: Jan. 14, 2020.

[14] T. Gomes, F. Salgado, A. Tavares, and J. Cabral, "CUTE Mote, A CUstomizable and Trustable End-device for the Internet of Things," *IEEE Sensors Journal*, vol. 17, pp. 1–1, 08 2017.

[15] Z. Wang, W. Li, and H. Dong, "Review on open source operating systems for internet of things," *Journal of Physics: Conference Series*, vol. 887, p. 012044, 08 2017.

[16] D. Oliveira, T. Gomes, and S. Pinto, "Towards a Green and Secure Architecture for Reconfigurable IoT End-Devices," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 335–336, 2018.

[17] T. Gomes, S. Pinto, T. Gomes, A. Tavares, and J. Cabral, "Towards an FPGA-based edge device for the Internet of Things," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pp. 1–4, 2015.

[18] R. Jedermann, T. Pötsch, and C. Lloyd, "Communication techniques and challenges for wireless food quality monitoring," *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, vol. 372, p. 20130304, 05 2014.

[19] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, "Operating Systems for Internet of Things

Low-End Devices: Analysis and Benchmarking," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10375–10383, 2019.

[20] Daniel Oliveira, Miguel Costa, Sandro Pinto, Tiago Gomes, "The Future of Low-End Motes in the Internet of Things: A Prospective Paper," pp. 1–21, 01 2020.

[21] M. Silva, A. Tavares, T. Gomes, and S. Pinto, "ChamelIoT: An Agnostic Operating System Framework for Reconfigurable IoT Devices," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 1291–1292, 2019.

[22] K. Kashiwagi, K. Saisho, and A. Fukuda, "Design and implementation of dynamically reconstructing system software," in *Proceedings 1996 Asia-Pacific Software Engineering Conference*, pp. 278–287, 1996.

[23] S. Nordstrom, L. Lindh, L. Johansson, and T. Skoglund, "Application specific real-time microkernel in hardware," in *14th IEEE-NPSS Real Time Conference, 2005.*, pp. 4 pp.–, 2005.

[24] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 79–80, 2013.

[25] K. Sharma, T. Suryakanthi, and T. Prasad, "Classification Of Heterogeneous Operating System," 09 2012.

[26] Shene, Ching-Kuang, "Multithreaded Programming Can Strengthen an Operating Systems Course," *Computer Science Education*, vol. 12, 10 2002.

[27] M. Farooq and T. Kunz, "Operating Systems for Wireless Sensor Networks: A Survey," *Sensors (Basel, Switzerland)*, vol. 11, pp. 5900–30, 12 2011.

[28] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, 2004.

[29] F. Javed, M. Afzal, M. Sharif, and B.-S. Kim, "Internet of Things (IoT) Operating Systems Support, Networking Technologies, Applications, and Challenges: A Comparative Review," *IEEE Communications Surveys Tutorials*, vol. PP, pp. 1–1, 03 2018.

[30] G. Schryen, "Security of Open Source and Closed Source Software: An Empirical Comparison of Published Vulnerabilities.," vol. 5, p. 387, 01 2009.

[31] A. K. Dwivedi and Meera Tiwari and Om Prakash Vyas, "Operating Systems for Tiny Networked Sensors: A Survey," 2009.

[32] "Contiki Documentation." [Online]. Available: http://contiki.sourceforge.net/docs/2.6/a01791.html.

[33] A. Dunkels, "Protothreads." [Online]. Available: http://dunkels.com/adam/pt/.

[34] T. Gomes, P. Lopes, J. Alves, P. Mestre, J. Cabral, J. L. Monteiro, and A. Tavares, "A modeling domain-specific language for IoT-enabled operating systems," in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pp. 3945–3950, 2017.

[35] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt, "Enabling large-scale storage in sensor networks with the Coffee file system," *2009 International Conference on Information Processing in Sensor Networks*, pp. 349–360, 2009.

[36] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-Level Sensor Network Simulation with COOJA," pp. 641–648, Nov 2006.

[37] H. Will, K. Schleiser, and J. Schiller, "A Real-Time Kernel for Wireless Sensor Networks Employed in Rescue Scenarios," pp. 834 – 841, 11 2009.

[38] Emmanuel Baccelli, Thomas Schmidt, and Matthias Wählisch, "RIOT: The friendly Operating System for the Internet of Things.." [Online]. Available: https://www.riot-os.org.

[39] E. Baccelli, O. Hahm, M. Wählisch, M. Günes, and T. Schmidt, "RIOT: One OS to Rule Them All in the IoT," 12 2012.

[40] Luo Yu-yan, "A Preliminary Discussion on Relationship between OSI RM and TCP/IP RM in Network Structural System," 2007.

[41] Y. Li, D. Li, W. Cui, and R. Zhang, "Research based on OSI model," in *2011 IEEE 3rd International Conference on Communication Software and Networks*, pp. 554–557, 2011.

[42] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, 1980.

[43] N. Muskinja, B. Tovornik, and M. Terbuc, "Use of TCP/IP protocol in industrial environment," in *IEEE International Conference on Industrial Technology, 2003*, vol. 2, pp. 896–900 Vol.2,

Dec 2003.

[44] A. Dunkels, J. Alonso, and T. Voigt, "Making TCP/IP viable for wireless sensor networks," 12 2003.

[45] Y. Mourtada, S. Wicker, and M. Swanson, "Statistical performance analysis of address-centric performance versus data-centric directed diffusion approach in wireless sensor networks," *Proc SPIE*, pp. 7–14, 07 2003.

[46] "Active IETF working groups." [Online]. Available: http://datatracker.ietf.org/wg/.

[47] "Approved IEEE Draft Amendment to IEEE Standard for Information Technology-Telecommunications and Information Exchange Between Systems-Part 15.4:Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANS): Amendment to Add Alternate Phy (Amendment of IEEE Std 802.15.4)," *IEEE Approved Std P802.15.4a/D7, Jan 2007*, 2007.

[48] Z. Sheng, S. Yang, Y. Yu, A. V. Vasilakos, J. A. Mccann, and K. K. Leung, "A survey on the ietf protocol suite for the internet of things: standards, challenges, and opportunities," *IEEE Wireless Communications*, vol. 20, pp. 91–98, December 2013.

[49] M. R. Palattella, N. Accettura, L. A. Grieco, G. Boggia, M. Dohler, and T. Engel, "On Optimal Scheduling in Duty-Cycled Industrial IoT Applications Using IEEE802.15.4e TSCH," *IEEE Sensors Journal*, vol. 13, no. 10, pp. 3655–3666, 2013.

[50] T. Gomes, F. Salgado, S. Pinto, J. Cabral, and A. Tavares, "Towards an FPGA-based network layer filter for the Internet of Things edge devices," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–4, 2016.

[51] T. Gomes, S. Pinto, F. Salgado, A. Tavares, and J. Cabral, "Building IEEE 802.15.4 Accelerators for Heterogeneous Wireless Sensor Nodes," *IEEE Sensors Letters*, vol. 1, no. 1, pp. 1–4, 2017.

[52] J. Higuera and J. Polo, "Understanding the IEEE 1451 standard in 6loWPAN sensor networks," in *2010 IEEE Sensors Applications Symposium (SAS)*, pp. 189–193, 2010.

[53] T. Gomes, F. Salgado, S. Pinto, J. Cabral, and A. Tavares, "A 6LoWPAN Accelerator for Internet of Things Endpoint Devices," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 371–377, 2018.

[54] P. Thubert, T. Winter, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, "RPL: IPv6 Routing Protocol for Low power and Lossy Networks," *IETF*, vol. RFC 6550, 03 2012.

[55] O. Iova, P. Picco, T. Istomin, and C. Kiraly, "RPL: The Routing Standard for the Internet of Things... Or Is It?," *IEEE Communications Magazine*, vol. 54, pp. 16–22, December 2016.

[56] "Transmission Control Protocol - Protocol Specification." [Online]. Available: https://tools.ietf.org/html/rfc793.

[57] A. Elnaggar, "TCP Vs. UDP," 10 2015.

[58] "Speed up machine-to-machine networking with UDP." [Online]. Available: https://www.embedded.com/speed-up-machine-to-machine-networking-with-udp/.

[59] "The Constrained Application Protocol (CoAP)." [Online]. Available: https://tools.ietf.org/html/rfc7252.

[60] A. Dunkels, "uIP - A Free Small TCP / IP Stack," 11 2001.

[61] T. B. Chandra, P. Verma, and A. K. Dwivedi, "Operating Systems for Internet of Things: A Comparative Study," pp. 1–6, 03 2016.

[62] S. Sciancalepore, G. Piro, G. Boggia, and L. Grieco, "Application of IEEE 802.15.4 Security Procedures in OpenWSN protocol stack," *IEEE Standards Education e-Magazine*, vol. 4, 12 2014.

[63] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister, "OpenWSN: A Standards-Based Low-Power Wireless Development Environment," *Wiley Transactions on Emerging Telecommunications Technologies*, vol. 23, p. 480–493, 08 2012.

[64] T. Chang, P. Tuset-Peiro, X. Vilajosana, and T. Watteyne, "OpenWSN OpenMote: Demo'ing a Complete Ecosystem for the Industrial Internet of Things," in *2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pp. 1–3, June 2016.

[65] T. Chang, T. Watteyne, and X. Vilajosana, "Competition: OpenWSN, a Development Environment for 6TiSCH," in *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, EWSN '19, (USA), p. 308–309, Junction

Publishing, 2019.

[66] "OSes for OpenWSN." [Online]. Available: https://openwsn.atlassian.net/wiki/spaces/OW/pages/5010229

[67] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister, "OpenWSN: A Standards-Based Low-Power Wireless Development Environment," *Wiley Transactions on Emerging Telecommunications Technologies*, vol. 23, p. 480–493, 08 2012.

[68] "Network Stack Diagram." [Online]. Available: https://openwsn-berkeley.github.io/firmware/index.html.

[69] "SmartRF06 Evaluation Board User's Guide," p. 45, 09 2012.

[70] "CC2538 Powerful Wireless Microcontroller System-On-Chip for 2.4-GHz IEEE 802.15.4, 6LoWPAN, and ZigBee® Applications," p. 34, 12 2012.

[71] "STM32L4 Series." [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32l4-series.html.

[72] W. Lamie and J. Carbone, "Measure your RTOS's real-time performance," 07 2020.

[73] "Measure your RTOS's real-time performance." [Online]. Available: https://www.embedded.com/measure-your-rtoss-real-time-performance/.