



**Ana Patrícia
Gonçalves dos
Santos**

**Protocolo de comunicações sem-fios em malha
para redes de iluminação pública
Street Lighting Mesh Network Protocol**



**Ana Patrícia
Gonçalves dos
Santos**

**Protocolo de comunicações sem-fios em malha
para redes de iluminação pública**

Street Lighting Mesh Network Protocol

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Doutor Paulo Pedreiras, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Paulo Bartolomeu, Diretor Técnico da Globaltronic - Electrónica e Telecomunicações, S.A.

o júri / the jury

presidente / president

Professor Doutor Alexandre Manuel Moutela Nunes da Mota
Professor Associado da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Luís Miguel Moreira Lino Ferreira
Professor Adjunto do Instituto Superior de Engenharia do Porto

Doutor Paulo Jorge de Campos Bartolomeu
Diretor Técnico da Globaltronic - Electrónica e Telecomunicações, S.A.

agradecimentos / acknowledgements

Cada uma das pessoas com as quais me cruzei durante estes anos desempenhou um papel imprescindível para a minha evolução e crescimento pessoal. Deste modo, no final deste ciclo cabe-me atribuir o devido mérito a todas as pessoas que de uma maneira ou de outra contribuíram para me mudar e me tornar quem hoje sou.

Ao meu professor e orientador Paulo Pedreiras pela excelente orientação e coordenação deste trabalho, pelos conselhos e pela permanente disponibilidade.

Ao meu co-orientador Paulo Bartolomeu pela sua incansável e inestimável orientação e por ser uma fonte constante de inspiração e motivação. Agradeço-lhe ainda a oportunidade que me ofereceu de desenvolver este trabalho na Globaltronic, onde consegui expandir os meus horizontes e conhecimentos de forma exponencial.

A todos os colaboradores da Globaltronic pelas trocas de ideias, aprendizagem, conselhos ou apenas conversas, opiniões e perspetivas de vida diferentes. Sem dúvida que aprendi bastante com cada um deles, o que me permitiu evoluir bastante como profissional.

A todos os meus professores e colegas de curso por todo o conhecimento transmitido ao longo destes anos.

A todos os meus amigos pelo apoio, amizade e pela força que me deram nos momentos mais críticos.

A toda a minha família com um destaque especial para os meus pais e irmãos, por todo o apoio e compreensão. Sem eles nada disto seria possível.

Palavras Chave

Internet das Coisas, Cidades Inteligentes, WSN, serviços Web embutidos, Contiki, RPL, 6LoWPAN, CoAP, OMA LWM2M

Resumo

A revolução digital do século 21 contribuiu para o surgimento da Internet das Coisas (IoT). Em breve trilhões de dispositivos embutidos usando o *Internet Protocol* (IP) serão parte integrante da Internet. De modo a suportar tal gama de endereços, um novo protocolo de Internet, chamado *Internet Protocol* versão 6 (IPv6) está a ser adoptado. O *IPv6 over Low power Wireless Personal Area Networks* (6LoWPAN) acelerou a integração das redes sem-fios de sensores na Internet. Ao mesmo tempo, o *Constrained Application Protocol* (CoAP) tornou possível fornecer funcionalidades de serviços Web RESTful a dispositivos com recursos limitados. Este trabalho baseia-se em experiências anteriores em redes de iluminação pública, para os quais um protocolo proprietário, elaborado pelo Lighting Living Lab, foi implementado e usado durante vários anos. O protocolo proprietário tem sido utilizado numa ampla gama de placas de controlo de iluminação. De modo a suportar aplicações heterogéneas com requisitos de comunicação mais exigentes além de melhorar o processo de desenvolvimento de aplicações, adaptou-se o Contiki OS à placa LED driver de 4 canais (4LD) da Globaltronic. Esta dissertação descreve o trabalho conduzido para adaptar o Contiki OS ao microprocessador MicrochipTM PIC24FJ128GA308 e apresenta uma solução baseada em IP para integrar sensores e atuadores em aplicações de iluminação inteligentes. Além da descrição da arquitetura e da implementação do sistema, este trabalho apresenta vários resultados que mostram que o desempenho do protocolo CoAP na placa 4LD é adequado para suportar serviços Web em redes de iluminação pública.

Keywords

Internet of Things, Smart Cities, embedded Web services, WSN, Contiki, RPL, 6LoWPAN, CoAP, OMA LWM2M

Abstract

The digital revolution of the 21st century contributed to stem the Internet of Things (IoT). Trillions of embedded devices using the Internet Protocol (IP), also called smart objects, will be an integral part of the Internet. In order to support such an extremely large address space, a new Internet Protocol, called Internet Protocol Version 6 (IPv6) is being adopted. The IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN) has accelerated the integration of WSNs into the Internet. At the same time, the Constrained Application Protocol (CoAP) has made it possible to provide resource constrained devices with RESTful Web services functionalities. This work builds upon previous experience in street lighting networks, for which a proprietary protocol, devised by the Lighting Living Lab, was implemented and used for several years. The proprietary protocol runs on a broad range of lighting control boards. In order to support heterogeneous applications with more demanding communication requirements and to improve the application development process, it was decided to port the Contiki OS to the four channel LED driver (4LD) board from Globaltronic. This thesis describes the work done to adapt the Contiki OS to support the MicrochipTM PIC24FJ128GA308 microprocessor and presents an IP based solution to integrate sensors and actuators in smart lighting applications. Besides detailing the system's architecture and implementation, this thesis presents multiple results showing that the performance of CoAP based resource retrievals in constrained nodes is adequate for supporting networking services in street lighting networks.

CONTENTS

CONTENTS	i
LIST OF FIGURES	v
LIST OF TABLES	ix
GLOSSARY	xi
1 INTRODUCTION	1
1.1 Purpose and Goals	1
1.2 Structure of this Thesis	3
2 KEY TECHNOLOGIES	5
2.1 Wireless Sensor Networks	5
2.1.1 Hardware Components	6
2.1.2 Networking	7
2.2 Operating Systems	8
2.2.1 Architecture	9
2.2.2 Programming model	10
2.2.3 Scheduling	10
2.3 Network Protocols	12
2.3.1 6LoWPAN	12
2.3.2 RPL - A Mesh Networking Solution	14
2.4 Application Protocols	17
2.4.1 CoAP	18
2.4.2 OMA Lightweight M2M	20
3 OPERATING SYSTEMS FOR WIRELESS SENSOR NETWORKS	25
3.1 Design Issues and Challenges	25
3.1.1 Restricted Resources	25
3.1.2 Portability	26
3.1.3 Customizability	26
3.1.4 Multitasking	26
3.1.5 Network Dynamics	27
3.1.6 Distributed Nature	27
3.2 Design Characteristics	27
3.2.1 Flexible Architecture	27

3.2.2	Efficient Programming Model and Scheduling	28
3.2.3	Clear Application Programming Interface	28
3.2.4	Reprogramming	29
3.2.5	Resource Management	30
3.2.6	Real Time Nature	30
3.3	Existing Operating Systems	30
3.3.1	Tiny OS	31
3.3.2	Contiki OS	32
3.3.3	Lite OS	33
3.3.4	Nano-RK	35
3.3.5	MANTIS	36
3.4	Evaluation of the Operating Systems	37
4	THE CONTIKI OPERATING SYSTEM	39
4.1	Brief Introduction	39
4.2	Main Features	39
4.3	Kernel and Processes	40
4.3.1	Events	42
4.3.2	Process Polling	43
4.3.3	The Process Scheduler	43
4.4	Protothreads	44
4.4.1	Protothreads in Processes	45
4.5	Preemptive Multi-threading	45
4.6	Memory Allocation	46
4.7	File Systems	48
4.8	The Dynamic Loader	49
4.9	Libraries	49
4.9.1	Timers	49
4.9.2	Leds API	50
4.9.3	The Serial I/O API	50
4.10	Communication	51
4.10.1	uIP Communication Stack	51
4.10.2	Rime Communication Stack	52
4.11	Global Overview	53
5	IMPLEMENTATION	55
5.1	Introduction	55
5.2	Hardware	56
5.2.1	The Giore Platform	56
5.2.2	The 4LD Platform	57
5.3	Porting the Hardware to Contiki OS	57
5.3.1	A General Port	58
5.3.2	Porting the Giore Platform	59
5.3.3	Porting the 4LD Platform	61
5.3.4	RFM69H Device Driver	63
5.4	The Network Stack	65
5.4.1	Physical, Framer, RDC and MAC Layers	65
5.4.2	Network Layer	66
5.4.3	Application Layer	67
5.5	Gateway	68

5.5.1	The Giore as Border Router	69
5.5.2	The SLIP tunnel	70
5.5.3	Functional Tests	71
5.6	Sensor Node	72
5.6.1	Experimental Setup using Erbium-CoAP and Copper	72
5.6.2	Experimental Setup using OMA LWM2M and Leshan Server	78
5.7	Development Tools	85
5.8	Difficulties during implementation	85
5.8.1	Stack Issues	85
6	EVALUATION OF THE IMPLEMENTATION	87
6.1	Memory Usage	87
6.2	Network: Performance Evaluation	88
6.3	CoAP transactions: Performance Evaluation	91
7	CONCLUSIONS	95
7.1	Conclusions	95
7.2	Future Work	96
	REFERENCES	97

LIST OF FIGURES

1.1	Wireless Sensor Networks Applications. Adapted from [3].	1
2.1	Wireless sensor network.	6
2.2	Hardware components of a sensor node. Adapted from [8].	6
2.3	WSN Topologies.	7
2.4	Conceptual view of an Operating System. Adapted from [12].	9
2.5	Header Compression Example. Adapted from [21].	13
2.6	Neighbor Discovery message exchange. Adapted from [22].	14
2.7	DODAG building process. The link quality means that on average, a packet sent on a specific path requires X transmissions before it reaches its destination. Adapted from [25].	16
2.8	CoAP frame format.	18
2.9	CoAP request/response model. Confirmable (left) and Non-confirmable (right). Adapted from [6].	19
2.10	The Lightweight M2M architecture with the LWM2M Client and the LWM2M Server. Taken from [33].	20
2.11	Standard Objects from LWM2M Technical Specification.	21
2.12	Abstract message flow example between a LWM2M Client and Server, the actual messages are mapped to CoAP requests and responses. Taken from [33]	22
2.13	IPSO Temperature and Light Control Objects Overview. Adapted from [35]. . .	23
3.1	The reprogramming flexibility and update cost, depending on the level of granularity. Adapted from [13].	30
3.2	Contiki architecture overview. Adapted from [46].	32
3.3	LiteOS architecture. Adapted from [41].	34
3.4	MANTIS architecture. Adapted from [41].	36
3.5	Operating Systems Summary.	37
4.1	Contiki execution contexts. Adapted from [56].	42
4.2	Synchronous event.	42
4.3	Asynchronous event.	43
4.4	Process scheduling and process polling in Contiki. Adapted from [57].	44
4.5	State chart of threads. Adapted from [59].	46
4.6	The managed memory allocator. Adapted from [60].	48
4.7	Contiki's communication model. Adapted from [45].	51
4.8	Contiki's operation overview. Adapted from [57].	53

5.1	The 4LD platform as led controller.	55
5.2	The Giore Board.	56
5.3	The 4LD Board.	57
5.4	Contiki's directory structure.	58
5.5	Giore Port main files.	60
5.6	PIC24 port main files.	62
5.7	4LD Platform port main files.	62
5.8	Radio device driver overview.	64
5.9	RFM69H packet fields.	64
5.10	Contiki OS network stack.	65
5.11	Network Stack used in the first experiment.	67
5.12	Network Stack used in the second experiment.	68
5.13	Overview of the gateway implemented in this thesis.	68
5.14	Terminal print after creating the SLIP tunnel.	70
5.15	Border router ping test.	71
5.16	4LD Nodes ping test.	71
5.17	Neighbors and routes defined in the border router.	72
5.18	Overview of the experimental setup using Erbium-CoAP and Copper Plugin. . .	73
5.19	CoAP resources implemented in the 4LD Server.	74
5.20	Server response to the GET action on the resource /.well-know/core.	75
5.21	Copper output from 4LD Node 3.	76
5.22	Copper output from 4LD Node 4. Since the 4LD LED channels are not mounted in this platform, the light_on and light_dim resources are not implemented. . .	76
5.23	Available actions in each resource.	77
5.24	GET message to request the sensor temperature value.	77
5.25	ACK message from the 4LD CoAP server with the temperature value.	78
5.26	Experimental setup using OMA LWM2M overview.	78
5.27	LWM2M implementation files.	79
5.28	LWM2M Client side implementation files.	80
5.29	Message sent from the 4LD LWM2M Client (Node 4) to make the registration in the Leshan Server.	81
5.30	Acknowledge sent from Leshan to the 4LD Client (Node 4) with the Registration ID.	81
5.31	Registration message sent from the 4LD LWM2M Client (Node 4) to the Leshan Server. The "j5DH4jpBKF" is the registration ID.	82
5.32	List of the connected clients in the Leshan Server.	82
5.33	List of objects defined in the 4LD Node 3.	83
5.34	List of resources defined in the IPSO Light Control object.	83
5.35	List of resources defined in the IPSO Temperature object.	84
5.36	List of resources defined in the LWM2M Server object.	84
5.37	List of resources defined in the Device object.	84
6.1	Firmware size comparison of the experimental setup using Erbium-CoAP and Copper.	88
6.2	Firmware size comparison of the experimental setup using OMA LWM2M and Leshan.	88
6.3	RTT and PLOSS evolution according to ICMP payload size for the experimental setup using Erbium-CoAP and Copper.	89
6.4	RTT and PLOSS evolution according to ICMP payload size for the experimental setup using OMA LWM2M and Leshan.	90

6.5	Response time of the CoAP resource requests for the experimental setup using Erbium-CoAP and Copper. The response time shown is the average result of 100 samples.	91
6.6	Total number of bytes needed to retrieve all the information from each resource. These results are for the experimental setup using Erbium-CoAP and Copper. .	92
6.7	Response time of the CoAP resource requests for the experimental setup using OMA LWM2M and Leshan. The response time shown is the average result of 100 samples.	92
6.8	Total number of bytes needed to retrieve all the information from each resource. These results are for the experimental setup using OMA LWM2M and Leshan. .	92

LIST OF TABLES

2.1	Comparison between event-based and thread-based. Adapted from [13].	10
4.1	Process control block fields description.	41
4.2	Process-specific protothread macros. Adapted from [56].	45
4.3	The memb API functions.	47
4.4	The malloc API functions.	47
4.5	The memm API functions.	48
4.6	The LEDs API functions.	50

GLOSSARY

WSN	Wireless Sensor Network	ICMPv6	Internet Control Message Protocol version 6
OS	Operating System	PC	Personal Computer
API	Application Programming Interface	CFS	Coffee File system
RF	Radio Frequency	CTK	Contiki Toolkit
TOS	TinyOS	ELF	Executable Linkable Format
CPU	Central Processing Unit	RFC	Request for Comments
MANTIS	Multimodal system for NeTworks of In-situ wireless Sensors	WPAN	Wireless Personal Area Network
MIPS	Million Instructions Per Second	IEEE	Institute of Electrical and Electronic Engineers
EEPROM	Electrically Erasable Programmable Read-Only Memory	OSI	Open Systems Interconnection
TCB	Thread Control Block	IETF	Internet Engineering Task Force
FIFO	First-In First-Out	CoAP	Constrained Application Protocol
SPI	Serial Peripheral Interface	HTTP	Hypertext Transfer Protocol
TYMO	DYMO protocol on TinyOS	URI	Uniform Resource Identifier
6LoWPAN	IPv6 over Low power Wireless Personal Area Networks	OMA	Open Mobile Alliance
IPv6	Internet Protocol version 6	LWM2M	Lightweight Machine-to-Machine
IPv4	Internet Protocol version 4	MCU	Microcontroller
IP	Internet Protocol	UART	Universal asynchronous receiver/transmitter
uIP	Micro Internet Protocol	ISR	Interrupt Service Routine
uIPv6	Micro Internet Protocol version 6	RDC	Radio Duty Cycle
TCP	Transmission Control Protocol	LLN	Low Power and Lossy Network
RPL	Routing Protocol for Low-Power and Lossy Networks	LoWPAN	Low-power Wireless Area Network
MAC	Media Access Control	MTU	Maximum Transfer Unit
USB	Universal Serial Bus	IPHC	IP Header Compression
UDP	User Datagram Protocol	NHC	Next Header Compression
ICMP	Internet Control Message Protocol	RA	Router Advertisement
		RS	Router Solicitation
		NS	Neighbor Solicitation

NA	Neighbor Advertisement	TLV	Type-Length-Value
ARO	Address Registration Option	IPSO	Internet Protocol Security Option
DAR	Duplicate Address Request	SLIP	Serial Line over Internet Protocol
DAC	Duplicate Address Confirmation	DAG	Directed Acyclic Graph
ROLL	Routing Over Low Power and Lossy Links	JSON	JavaScript Object Notation
DODAG	Destination Oriented Directed Acyclic Graph	DTLS	Datagram Transport Layer Security
DIO	DODAG Information Object	RTT	Round Trip Time
DIS	DODAG Information Solicitation	PLOSS	Packet Loss
DAO	DODAG Advertisement Object	RAM	Random Access Memory
OF	Objective Function	ROM	Read Only Memory
REST	Representational State Transfer	QoS	Quality of Service
XML	eXtensible Markup Language	IoT	Internet of Things

INTRODUCTION

1.1 PURPOSE AND GOALS

In the recent years, there has been a significant increase in the use of Wireless Sensor Networks (WSNs) in industry [1], [2]. The spectrum of applications is very vast, as can be seen in Figure 1.1.

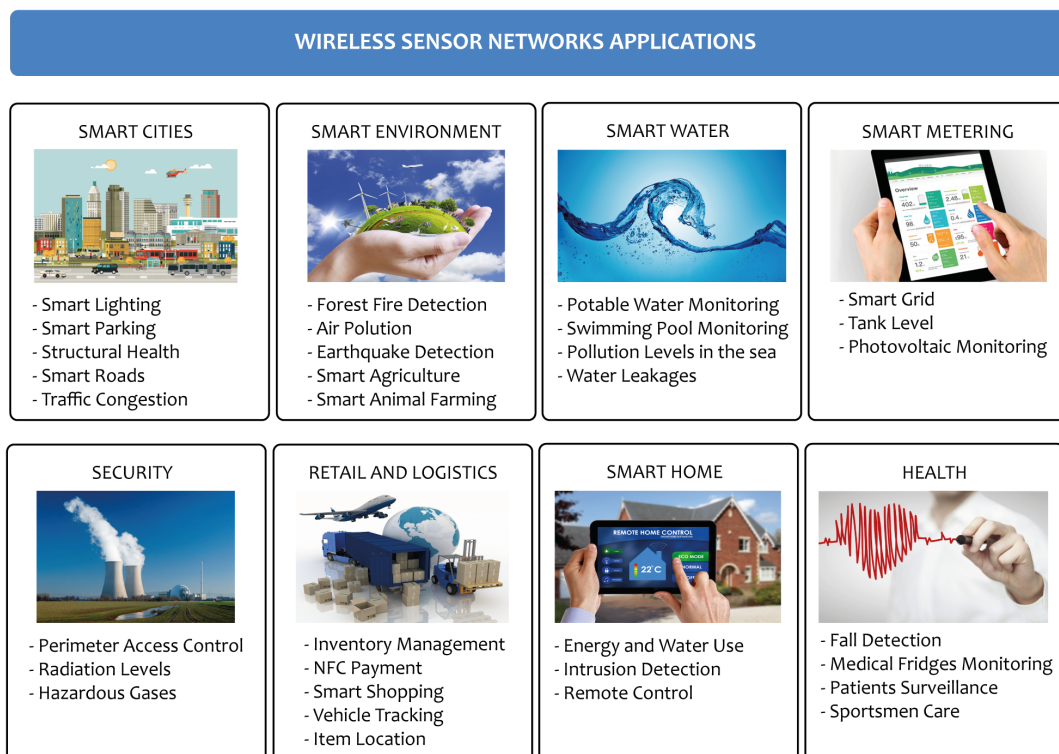


Figure 1.1: Wireless Sensor Networks Applications. Adapted from [3].

In addition to the application domains referred to in Figure 1.1, many others can be found, as shown in [4] and [5]. The numerous areas where the WSNs are applied has pushed the research and development in embedded operating systems to support heterogeneous applications. In fact, an embedded OS will turn the applications hardware independent, and will allow the reuse of communication stacks, rather than developing one from scratch, which would undoubtedly be a very consuming task. These features will enable very significant gains across the applications' development process.

This work was developed in partnership with Globaltronic S.A.. Globaltronic has expertise in various areas and has developed a range of products for control and lighting management. One of these products is the platform 4LD, which has been used to control street and industrial lighting networks. The communication between nodes of this network is made using a proprietary RF protocol. In order to support applications with more demanding communication requirements and to improve the application development process, it was decided to adapt the Contiki Operating System (OS) to the 4LD platform. Chapter 3 discusses why we have chosen this OS for our application.

The 4LD platform is based on the PIC24FJ128GA308 processor, thus this dissertation describes the work carried on to adapt the Contiki operating system to support this microprocessor. Afterwards, the goal was to connect the nodes to a Wireless Mesh Network, so that they can be reached from the Internet. Nowadays IPv4 is the main protocol used on the Internet, however a transition to IPv6 is being gradually implemented. The reason that led to this transition is related to the limited addressing capabilities of IPv4. IPv4 has only 2^{32} possible addresses. Given that in the future it is estimated that trillions of devices will be connected to the Internet, it would be completely impossible to use the IPv4 for this purpose. IPv6 solves this problem by enlarging the address space to 2^{128} possible addresses. This should make it possible for every device to have a unique address.

The final goal of this thesis is to actually send useful data using IP over the wireless mesh network. There are several of application layer technologies that can be useful for this purpose. One of these technologies is CoAP, or Constrained Application Protocol. CoAP is a Web application layer protocol that is specially designed for devices with constrained resources. CoAP is quite similar with the HTTP protocol, widely used on the Internet nowadays. The main difference is that CoAP, as it was designed taking into account limited devices, it is far more compact when compared to HTTP. However, both use URIs to locate resources, they offer the possibility of adding a Content Type header that describes the format of data and both use mechanisms to ensure that the message is reliably transferred [6]. HTTP and CoAP also share a common set of request methods: GET, POST, PUT and DELETE. All mentioned features make CoAP easy to understand and integrate into the current architecture of the Web.

This work is aimed at evaluating the possibility of supporting smart light applications in embedded systems connected by wireless communications over IP. Such embedded devices are characterized by severe resource constraints, but will benefit significantly in having an operating system supporting the key functionalities. Several aspects of the implementation should also be examined, such as memory usage and network performance. An evaluation of the CoAP requests, in terms of response time, will also be performed.

1.2 STRUCTURE OF THIS THESIS

In Chapter 1 we contextualize the motivations that led to the development of this thesis and also the objectives proposed in this work.

In Chapter 2 we present the key concepts pertaining to the technical field in which this thesis is developed. Concepts like Wireless Sensor Networks and Operating Systems are introduced, since they will be very useful in the following chapters. The protocols used in each layer of the network stack implemented are also introduced.

In Chapter 3 we present an overview of several different operating systems for WSNs. First, the challenges and design issues that may affect the design of an operating system for WSNs are presented. The main design characteristics of the OS are also considered. Then, a small description of the different operating systems is provided, taking into account these design characteristics. After looking at the various operating systems available in the literature, a comparison was made and it was decided to use the Contiki OS.

In Chapter 4 we describe the Contiki operating system in more detail, demonstrating their unique characteristics and the advantages that such characteristics brought to Wireless Sensor Networks.

In Chapter 5 our implementation is presented from a hardware and software perspective. We also discuss the problems encountered during the implementation phase.

In Chapter 6 we evaluate the implementation. The memory usage and the network performance are evaluated in terms of round trip time and packet loss finally, the response time for the different implemented CoAP resources is also assessed.

In Chapter 7 the results of this thesis are summarised, and possible future work is discussed.

KEY TECHNOLOGIES

The following section aims to present the basic concepts related to the technologies and protocols used during this work. Firstly, the concepts related to Wireless Sensor Networks and Operating Systems will be introduced. Secondly, the protocols that will be used to implement the network stack of our lighting mesh network will be addressed. Protocols used in the network layer, such as 6LoWPAN and RPL, and also the CoAP and OMA LWM2M, used in the application layer, will be discussed. The key technologies presented in this section are crucial for a proper understanding of the implementation developed in this thesis.

2.1 WIRELESS SENSOR NETWORKS

The smart environments are undoubtedly the next evolutionary development step for cities, utilities, industrial, home, transportation and agriculture [7]. In the future all of these smart devices will be interconnected, creating a network on a global scale. Thus, the interest in WSNs is steadily growing. Wireless Sensor Networks (Figure 2.1) consist of several wireless devices, extremely efficient in terms of energy usage, and capable of transmitting sensor data using low-power and low-bandwidth links [7].

A WSN generally encompasses the following key elements:

- **Sensors:** elements that have the capability to transduce (or detect) a given characteristic on the environment, providing an output in the electrical form;
- **Nodes:** elements that collect the information of their sensors and sends it to the base station;
- **Gateway:** element that bridges the communications between the sensor network and the Internet;
- **Base station:** element that gathers all the data that comes from the sensor nodes and processes it according to the application requirements.

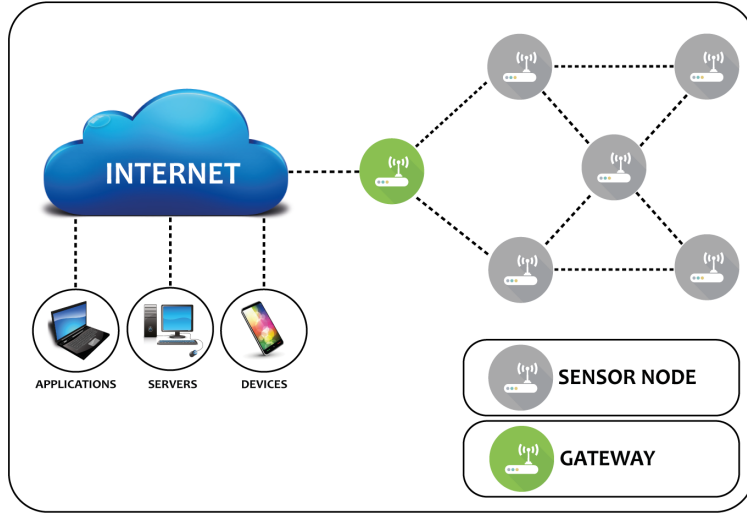


Figure 2.1: Wireless sensor network.

2.1.1 HARDWARE COMPONENTS

Each sensor node of a WSN is defined both by its physical construction (the hardware) and by its behaviour (the software). The typical sensor node encompasses the following hardware components (Figure 2.2) [7]:

- **Communication device:** Typically is a radio transceiver responsible for providing communication capabilities to the node;
- **Microcontroller:** Runs the software of the smart object. This software will define the behavior of the sensor node;
- **Set of sensors or actuators:** Responsible for providing the node a way to interact with their surroundings;
- **Power source:** A crucial element of the sensor node. Without it his electrical circuits will be useless.

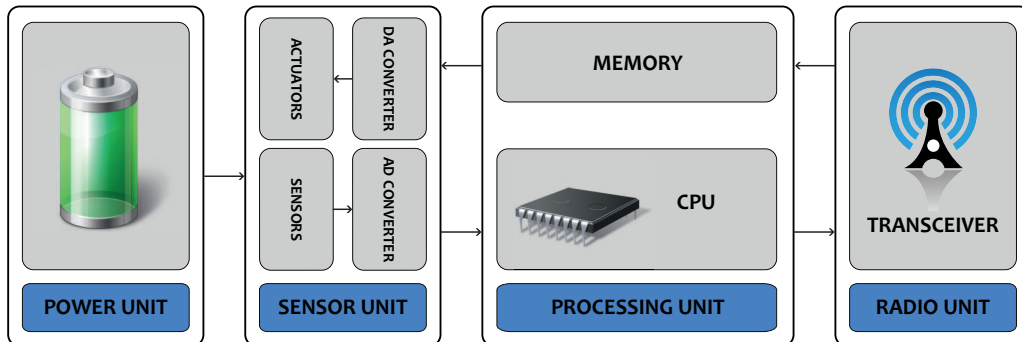


Figure 2.2: Hardware components of a sensor node. Adapted from [8].

2.1.2 NETWORKING

Two main networking topologies are used in the WSNs: star and mesh topologies. Star networks are constituted by an aggregation of point-to-point links, with a master node that manages a specific number of slave nodes [9]. This master node acts as the root for all the upstream communications (Figure 2.3). If one of the slave nodes needs to communicate, it must forward the message to the root node. One of the disadvantages of the star topology is his lack of robustness. Given that all network packets pass through the master node, this node becomes a single point of failure. If he fails, the entire sub-network will fail.

In a mesh topology (Figure 2.3), each node has several possible routes for a given node, which provides the most flexibility and robustness. Most practical mesh networks use a type of mesh with peer-to-peer communication links that support routing. Messages cross the network using a multi-hop routing algorithm that can be optimized for the lowest latency, lowest power consumption or any other desired metric [9]. Given that each of the sensor nodes must have a routing table for the remaining nodes of the mesh network, the memory requirements and processing overhead required in each node are higher in mesh topologies.

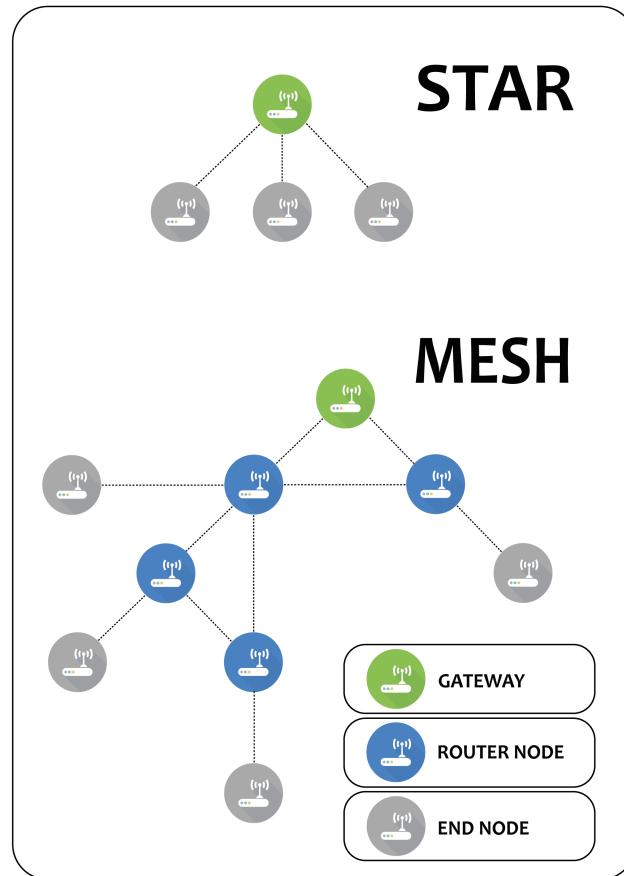


Figure 2.3: WSN Topologies.

A wide variety of proprietary wireless low-power networking technologies have been released since the 1990s until early 2000s. However, it was only in 2003 that the Institute of Electrical and Electronic Engineers (IEEE) launched the first low-power wireless personal area network (WPAN) standard: the IEEE 802.15.4. This standard defined the Physical and Media Access Control (MAC) layers from the

OSI model [10]. Based on that standard, the ZigBee Alliance provided commercial wireless embedded networking solutions for various areas [11].

The majority of the WSN solutions that currently exist do not provide support for IP. To provide interoperability between these WSNs and external networks it is necessary to use specially designed gateways. Furthermore, the protocol used over the Internet these days, IPv4, can not be adopted due to being unable to address such a wide range of smart devices. With the emergence of IPv6, the address space available increased considerably in order to support billions of embedded devices. However, the complexity of providing IPv6 for devices with highly constrained memory and processing power has become a great challenge. The IETF has assigned two different working groups to integrate IPv6 in WSN devices:

- **6LoWPAN:** Adaptation layer for IPv6 packets on IEEE 802.15.4 MAC messages;
- **RPL:** Routing protocol for low-power and lossy networks, that provides efficient routing mechanisms in terms of energy.

In the scope on this work, it is beneficial to use open-source software, since we want to be able to adapt the software to our specific needs without any additional costs. Currently, there are a variety of open-source implementations for WSNs. After an analysis of all existing implementations, we highlight the operating systems specially built for embedded systems, such as TinyOS and Contiki. These operating systems are open-source and provide implementations of the IEEE 802.15.4, IETF 6LoWPAN and RPL routing for several different memory and power constrained devices. Using an open-source operating system to implement our network will definitely offer more freedom in developing solutions for specific networking requirements.

The basic functionality of an operating system is to provide an abstraction layer that hides the low-level details of the sensor node. This layer will provide a clear interface to the application(s). An OS is also responsible for offering low-level services such as processor management, scheduling policies, multi-threading, multitasking, etc. These services are quite similar to those presented in traditional operating systems. However, their implementation in WSN is a non-trivial problem, due to the resource constraints of the sensors [7].

Chapter 3 discusses the challenges and issues that may affect the design of an operating system for WSNs. In section 2.2 some important concepts related to operating systems, like programming models, architectures and scheduling policies are introduced.

2.2 OPERATING SYSTEMS

An operating system aims to provide its users an environment where they can execute programs conveniently and efficiently. In technical terms, it is a software which manages hardware [12]. Therefore, an operating system is responsible for the allocation of resources and services, such as memory, processors, devices and information.

Operating Systems generally consist of several parts. The principal ones are [12]:

- The Kernel, which is the "core" of the OS. It is responsible for running programs, and ensures that these programs access the hardware safely. As there are several programs to dispute the resources not always abundant, the kernel is also responsible for deciding which program has

more priority to run and how long should it run. This is called scheduling. Scheduling concepts are introduced in Subsection 2.2.3;

- The Libraries, which provide a set of functions that can be used by applications;
- The Drivers, which are responsible for controlling external hardware.

Figure 2.4 represents the conceptual view of an Operating System.

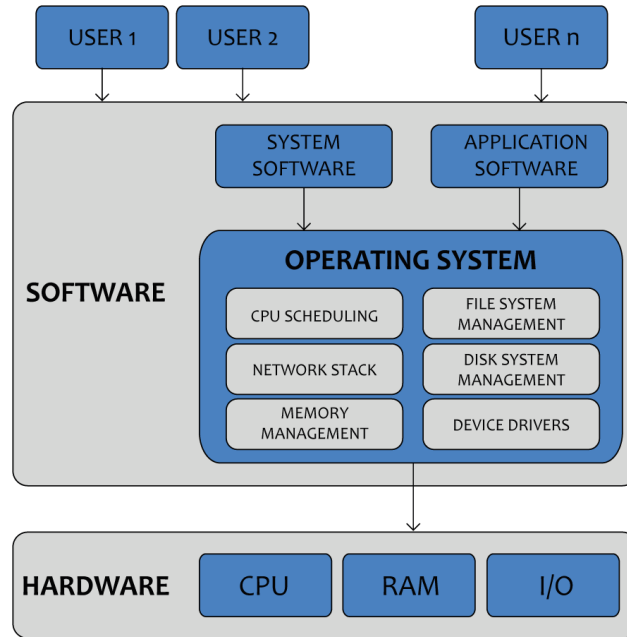


Figure 2.4: Conceptual view of an Operating System. Adapted from [12].

2.2.1 ARCHITECTURE

The organization of an OS constitutes its structure. The architecture of an OS will not only influence the size of the OS kernel, but also the way it provides services to the programs. Some of the well known OS architectures are [13]:

- Monolithic - Application + Necessary OS components = Single system image;
- Modular - Application and OS are built as a set of interacting modules;
- Virtual Machine - Application as a set of static and dynamic components = Network wide single system image.

Monolithic: This type of architecture differs from the others because it defines a high abstraction layer over the computer hardware [12]. All operating system services such as process management, concurrency and memory management, are implemented using a set of primitives/system calls. This architecture allows the grouping of all the required services in a single system image, thus resulting in a smaller OS memory footprint. However, a monolithic OS is hard to understand and modify, is unreliable and also difficult to maintain [13].

Modular: In a modular architecture the operating system components are described as communicating object-oriented modules. The kernel only has the necessary core components to start itself and the ability to load modules. The core module is the only module that is always in memory. Whenever any other additional modules are required, the module loader is responsible for loading the correct module. The main advantage is the possibility of runtime reconfiguration, however, there is an overhead in loading and unloading modules [13].

Virtual Machine: Virtual machines run inside user programs on top of another OSs. The main advantage is its portability, however, the system performance is typically poor [12].

2.2.2 PROGRAMMING MODEL

Traditionally, programming models for concurrent processes can be of two types:

- Thread-based: Each process is defined by implementing a thread-specific method. The execution state of the processes threads is maintained by an associated thread stack;
- Event-based: Each process is implemented by event handlers which are called from inside an event loop. The execution state of the process is stored within an associated record or object.

Thread-based models are easier to use, but when it comes to performance, they are less efficient, due to context switches, memory consumption, etc. [14]. Event-based models are more efficient, but hard to use in complex designs [15]. The comparison between the two can be found in Table 2.1.

Event-based	Thread-based
Computation is handled by event handlers	Computation is divided between threads
No stack overhead	Context switch overhead
Used when applications require efficiency	Used when applications require flexibility
Allows high concurrency	Not well suited for concurrency

Table 2.1: Comparison between event-based and thread-based. Adapted from [13].

2.2.3 SCHEDULING

One important part of an Operating System's kernel is the CPU scheduler. The scheduler is responsible for deciding when and for how long a process is allowed to execute. Since the CPU has to offer the illusion of concurrent processing, the scheduler must ensure that performance is not hindered and that processes run according to a set of policies [12]. The scheduling policies are defined in scheduling algorithms. These fall into two categories: **preemptive** and **nonpreemptive**.

Tasks are usually assigned with priorities and sometimes it may be necessary to perform a certain task that has a higher priority before another task already running. In preemptive scheduling, the running task is interrupted for some time and resumed later when the higher priority task has finished its execution [12]. In the case of nonpreemptive scheduling, when a process enters the running state,

it is not suspended until it finishes its service time. So, basically, in this type of scheduling, tasks of lower priority can block tasks of higher priority [12].

A scheduling algorithm, to be useful, must enforce the system policies and allocate the CPU to ready processes. Different policies can be used to select which processes to execute.

Preemptive algorithms:

- Round-robin;
- Priority scheduling;
- Shortest remaining time next;
- Shortest process next;
- Multiple queues;
- Guaranteed scheduling;
- Lottery scheduling;

In round-robin scheduling a certain amount of time to run is allocated to each process and a linked list contains all the ready processes. Then, the CPU is assigned to the process that is at the top of the list. This process will run during the time that was allocated to it and when the time is up the process goes to the back of the list [12].

The only critical issue with this algorithm is that it does not differentiate between compute-based operations and I/O-based ones. The I/O operations clearly must have a higher priority, because these operations can be critical to the proper functioning of the system. The solution is to create multiple queues for each priority level and then the scheduler executes them in a round-robin fashion, giving priority to those in the higher priority queues [12].

Nonpreemptive algorithms:

- First-come First-served;
- Shortest job first;

In First-come First-served scheduling, a single queue contains all the ready processes [12]. When it is done, the next process on the queue runs.

In systems where temporal constraints are critical, such as real-time systems, the tasks must execute before deadlines, or it may lead to catastrophic situations. In these cases proper scheduling algorithms must be used (e.g., Earliest Deadline First or Rate Monotonic, which support preemption).

The Rate Monotonic is a fixed priority scheduling algorithm that consists of assigning the highest priority to the highest frequency tasks in the system, and lowest priority to the lowest frequency tasks. Logically, the scheduler will always choose to execute the task with the highest priority. Using this algorithm the behavior of the system can be analyzed *apriori*, since the period and computational time required by each of the tasks is specified. One problem with the rate monotonic is that the schedulable bound is less than 100 % and also does not support dynamically changing periods [16]. The disadvantages associated to this algorithm encourage the use of dynamic priority algorithms, like Earliest Deadline First.

The Earliest Deadline First algorithm uses the deadline of a task as its priority. The task with the earliest absolute deadline has the highest priority, while the task with the latest absolute deadline has the lowest priority. One advantage of this algorithm is that the schedulable bound is 100 % for all task sets and because priorities are dynamic, the periods of tasks can be changed at any time [16].

2.3 NETWORK PROTOCOLS

2.3.1 6LOWPAN

The kind of devices used in this thesis, especially the 4LD, do not allow the transmission of very large packets, due to limitations in the radio transceiver FIFO, not to mention the issues related to the processing overhead and insufficient memory to buffering these packages. Given that the Maximum Transfer Unit (MTU) of an IPv6 packet is 1280 bytes, these devices cannot use this protocol directly. Therefore, we need an adaptation layer in order to use the IPv6 protocol in our implementation. For this purpose, we used the 6LoWPAN protocol. 6LoWPAN is an acronym that stands for IPv6 over Low power Wireless Personal Area Networks (WPAN). 6LoWPAN enables the use of IPv6 in Low Power and Lossy Networks (LLNs), such as those based on the IEEE 802.15.4 standard [17].

The 6LoWPAN architecture is made up of low-power wireless area networks (LoWPANs), which are connected to other IP networks through border routers. The border router plays an important role as it routes traffic in and out of the LoWPAN, while handling 6LoWPAN compression and Neighbor Discovery. In Section 5.5 the border router importance as part of the gateway node will be discussed. Each node is identified by a unique IPv6 address, and is capable of sending and receiving IPv6 packets. Typically, LoWPAN nodes support ICMPv6 traffic and employ the User Datagram Protocol (UDP) as the transport protocol.

Taking into account that the MTU size in IPv6 networks is 1280 bytes and that the maximum size for a IEEE 802.15.4 packet is just 127 bytes, header compression and message fragmentation must be performed in order to adapt IPv6 communications to IEEE 802.15.4 devices [18].

Fragmentation: Packet fragmentation is possible using a subheader in all fragments, encompassing fields such as Datagram Tag and Datagram offset. The Datagram Tag is used to identify the set of unfragmented payload the fragments belong and the Datagram offset identifies the offset of the fragmented packet within the unfragmented payload. However, the applications should avoid the transmission of big packets that require fragmentation, due to performance issues. Considering that these networks are lossy networks, if a fragment of a packet is lost, the whole packet will have to be retransmitted [19].

Header Compression: IPv6 addresses are composed by 128 bits. The first 64 bits are the prefix, common to all devices on the network, and the remaining 64 are the interface ID. The RFC4944 introduced the concept of IPv6 header compression (HC1) and UDP header compression (HC2) [18]. The prefix is known to all devices, therefore can be omitted. The interface IDs can also be omitted for link-local communication. An UDP/IPv6 header is usually composed of 48 bytes. If we use both HC1 and HC2 mechanisms, the header can be compressed to only 7 bytes, considering the case where a datagram is sent inside the 6LoWPAN network using the 16-bit addresses. However, outside of the unicast link-local scope, the HC1 and HC2 mechanisms cannot perform an efficient header compression. In a link-local multicast IPv6 packet the full destination address must be included, imposing a 23-byte header in the best situation. When communicating with nodes from an external network, the IPv6 source address prefix and full IPv6 destination address must be carried inline, resulting in a 31-bytes long header [18].

To address the previously reported problem, RFC6282 has introduced new header compression mechanisms [20]:

- IP Header compression (IPHC) - is used to efficiently compress fields in the IPv6 header such as Traffic Class, Flow Label and Hop Limit, using shared context information to omit the prefix from IPv6 addresses;
- Next Header Compression (NHC) - uses a similar mechanism to compress UDP headers, however it allows future definitions of arbitrary next header compressions.

Using the mechanisms introduced in the RFC6282, as can be seen in Figure 2.5, the UDP/IPv6 headers can be compressed to 6 bytes in the link-local scope, 7 bytes to known multicast addresses and 10 bytes with global addresses [21].

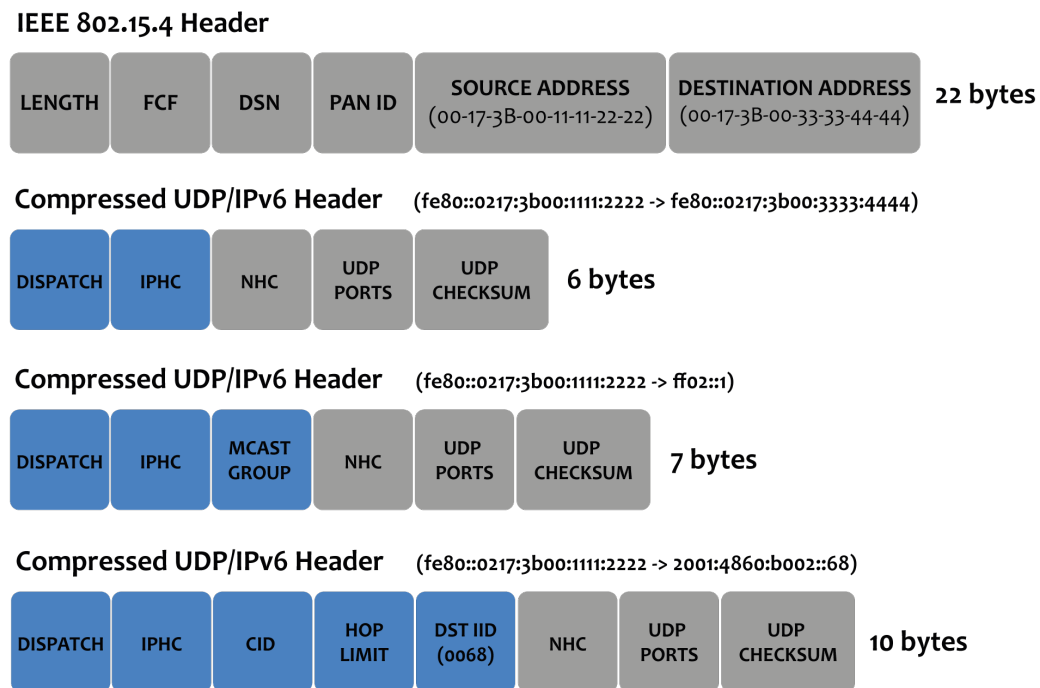


Figure 2.5: Header Compression Example. Adapted from [21].

Network Autoconfiguration: 6LoWPAN also allows network automatic configuration, using the neighbor discovery protocol [21]. As in normal IPv6 networks, there are several types of messages that are exchanged to perform this automatic configuration. Examples are:

- Router Advertisement (RA) - are sent to automatically propagate router information across the 6LoWPAN network;
- Router Solicitation (RS) - are sent by end nodes to locate a router in the network;
- Neighbor Advertisement (NA) - are used by nodes to respond to a NS message;
- Neighbor Solicitation (NS) - are sent by end nodes to determine the link layer address of a neighbor, or to verify that a neighbor is still reachable via a cached link layer address.

End Nodes can also send Neighbor Solicitation messages with Address Registration Option (ARO) to register their addresses to routers. 6LoWPAN routers may also send a specific type of NS messages

to border routers to perform Duplicate Address Detection, with the use of the ICMP messages DAR (Duplicate Address Request) and DAC (Duplicate Address Confirmation) [22]. In Figure 2.6 a typical neighbor discovery message exchange is presented, between the WSN nodes, the border router and an IPv6 base station.

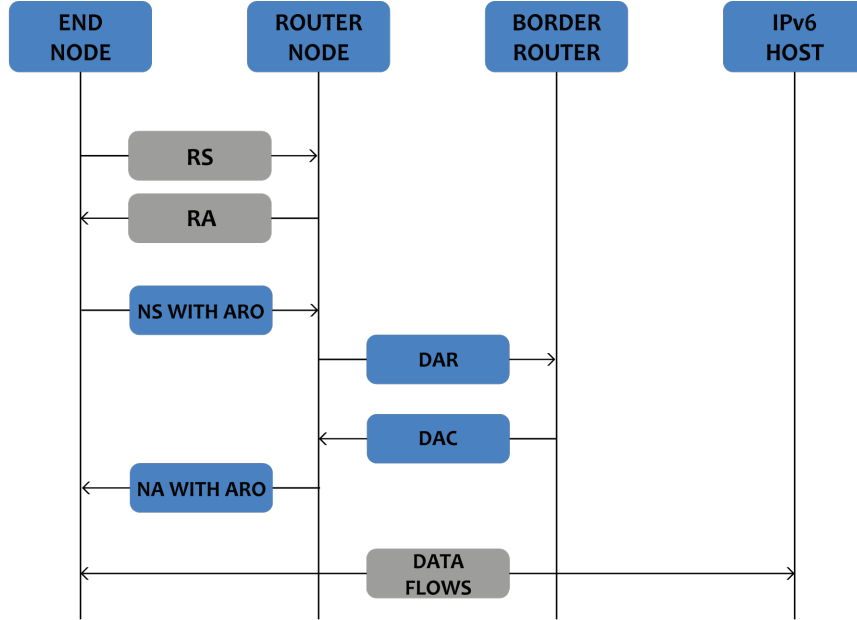


Figure 2.6: Neighbor Discovery message exchange. Adapted from [22].

Routing: There are two distinct options to be considered, when talking about routing in a 6LoWPAN network: mesh-under and route-over. Mesh-under techniques do not perform any IP routing within the LoWPAN. They typically use the layer 2 functions, such as IEEE 802.15.4, to perform the multi-hop forwarding [23]. In the route-over mechanisms, the routing functions are performed on the network layer, with every node acting as an IP router, and each link-layer hop as a single IP hop. The IETF did not develop any mesh-under routing protocols, but 6LoWPAN supports several route-over routing protocols, such as mobile ad-hoc network protocols like AODV and DYMO [19]. However, these protocols are not optimized to operate on LLNs, thus IETF created the workgroup ROLL (Routing Over Low Power and Lossy Links) to deal with this drawback. As an answer, the RPL protocol emerged [24]. In the following subsection this routing protocol will be introduced.

2.3.2 RPL - A MESH NETWORKING SOLUTION

RPL is a Distance Vector IPv6 routing protocol for LLNs that specifies how to build a Destination Oriented Directed Acyclic Graph (DODAG) using an Objective Function (OF) and a set of metrics/-constraints. The objective function will compute the best path, based on the combination of those metrics/-constraints. Since a single mesh network may need to carry traffic with different requirements in terms of path quality, there could be several objective functions in operation on the same mesh network [7]. For example, several DODAGs may be used with the objective to:

- Find paths with best expected transmissions values (metric) and avoid non-encrypted links (constraint);
- Find the best path in terms of latency (metric) while avoiding battery-operated nodes (constraint).

The OF will provide some rules for the formation of the DODAG, such as the number of parents, backup parents, etc. The DODAG building process is done using ICMPv6 control messages, such as:

- DIO - DODAG Information Object
- DIS - DODAG Information Solicitation
- DAO - DODAG Advertisement Object

The DODAG building process starts always by designating one node as the root node. In our specific experimental setups the root node is the border router. Therefore, the border router will be responsible for determining the configuration parameters for the network. These parameters will be packed into a DIO message, which is then used to disseminate the information in the network. The DIO messages contain many options that can be configured to tailor the network configuration to the application's requirements. The compulsory information contained in a DIO mainly comprises[24]:

- RPLInstanceID - unique identifier of an RPL Instance in a network;
- The DODAGID - unique identifier of an DODAG in an RPL Instance;
- The current DODAG version number;
- The node's rank - the logical distance from the root node within the DODAG.

DODAG Building Process: During the DODAG building process, each node is required to select parent nodes from its neighbors and must calculate its rank. The rank of each node must be larger than the rank of all its parents, so we can prevent the formation of loops in the routing structure. Thus, if we traverse the DODAG from the root node to the end nodes, the node's rank is monotonically increasing [24]. It is important to mention that rank is not necessarily related to the physical distance, nor to the distance in hops between a node and the root node, but a metric determining a node's desirability as a next hop on a route to the root node. A node's rank is calculated based on the OF, which is specified according to the DODAG's application goals [25]. The various steps of the graph building process are represented in Figure 2.7.

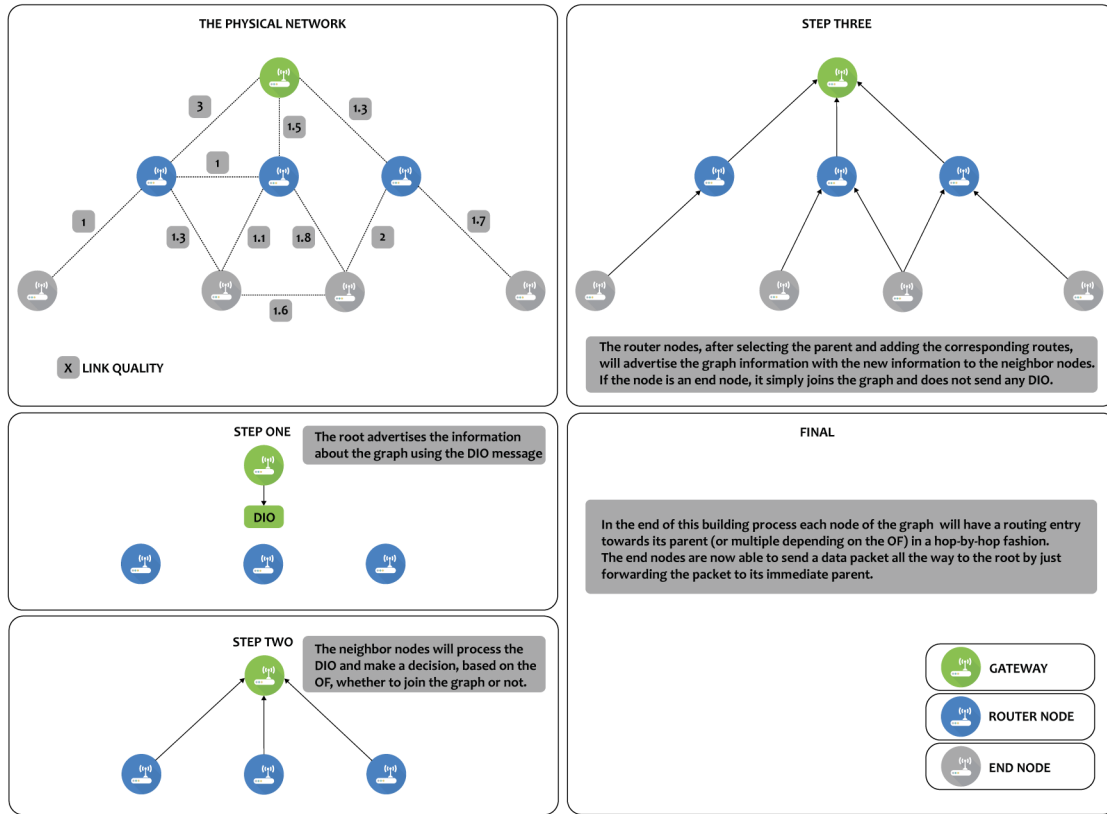


Figure 2.7: DODAG building process. The link quality means that on average, a packet sent on a specific path requires X transmissions before it reaches its destination. Adapted from [25].

Types of communication: There are three types of communication in RPL networks:

- **Multipoint-to-point (MP2P)** - The model presented before is a type of MP2P forwarding model, where each end node has connectivity towards the root node of the DODAG graph. Each node forwards every packet to its corresponding parent node. This forwarding process is performed until the packet reaches the root node. This is also referred to as UPWARD routing.
- **Point-to-multipoint** - In contrast to what happens with MP2P, where the traffic is forwarded from the end nodes to the root node, there is also a need for traffic to flow in the opposite or "down" direction. This traffic could be originated, for example, if the root node wants to send a message to an end node. This will require a routing table to be built at every node. In order to support traffic in the "down" direction, the DAO messages are used. These messages advertise prefix reachability towards the end nodes. Therefore, these messages will usually carry: the prefix information, the valid lifetime and information about the distance of the prefix [25]. As each node joins the graph it will send a DAO message to its parent node. Each node, upon receiving this message, will process the prefix information and add a new route to the routing table. Once this information reaches the root node, an entire path to the prefix is setup. This mode is called the storing mode of operation. In this mode, the DODAG nodes should have available memory to store the routing tables, otherwise it is not possible to support traffic in this direction [24]. Since these nodes typically are resource constrained devices, they do not support

the maintenance of large routing tables. To resolve this issue RPL also supports another mode called non-storing mode where the nodes do not need to store any routes. Instead, only the root node will compute and maintain a routing table to each node in the DODAG, based on DAO messages received from the remaining nodes. So, when the root wants to send a packet to a specific node, it will include the route in the source routing header and will send it to the next child node. Each child node will examine that field to know the next hop. This process will be repeated until the packet reaches its destination [24]. This mode of operation is more efficient in memory constrained devices, however it has the tradeoff of having a larger overhead, due to an increased packet size, which will use more power, processor resources and bandwidth [25].

- **Point-to-Point** - The data packets in RPL can also be forwarded from any node to any other one in the graph. In this case, the packet travels 'up' to the root node and then it is forwarded in the "down" direction to the destination node.

Topology repair: RPL implements two mechanisms of topology repair: local and global repair. If a node detects that one of his neighbors has failed and the node has no route towards the root node, a local repair is performed in order to find an alternate route. A local repair has no implications on the global topology. However, successive repairs may lead to a less efficient global topology and the root node may have to perform a global repair, reshaping the entire tree [24].

The trickle timer: In LLNs the network may be composed of battery-operated devices that must save energy. Therefore, it becomes imperative to limit the control traffic in the network. Most of the routing protocols use periodic keepalives to keep the routing tables up to date. This would be costly in LLNs, since the energy resources are scarce. To avoid energy waste, RPL uses an adaptive timer mechanism called the "trickle timer" [24]. This timer will control the sending rate of DIO messages. In the beginning, when building the DODAG this rate will be higher, so more DIO messages are sent. As the network stabilizes, the interval of the trickle timer will increase, which results in fewer DIO messages being sent in the network. If inconsistencies are detected, for example when a node joins the network or moves within the network, the nodes reset the trickle timer and will send DIOs messages more often. The frequency is increased only in the vicinity where the inconsistency is detected. So, using this mechanism the frequency of the DIO messages will depend on the stability of the network. As the network becomes stable, the number of RPL messages will gradually decrease. One of the main benefits of the trickle timer is that it does not require complicated code and it is also straightforward to implement. This is particularly necessary taking into account the usually resource constrained devices, that are comprised in these kind of networks.

2.4 APPLICATION PROTOCOLS

As stated before, the final goal of this thesis is to actually send useful data using IP over the wireless media. One of the key advantages of IP based networking in WSNs is to enable the use of standard Web services without the need to add special designed gateways. The idea is to integrate these smart objects into the Web on top of Representational State Transfer (REST) architectures [26]. In REST architectures each Web resource is identified by a Universal Resource Identifier (URI). These resources are manipulated using an application protocol. REST is not tied to a particular application

protocol, however, the vast majority of REST architectures nowadays use the HTTP protocol. The HTTP protocol is considered to be a heavy-weight resource representation format, designed to be used with devices with abundant resources. For resource constrained devices, like the ones used in this work, the HTTP protocol is not well suited [27]. Thus, there is a need for an application protocol integrated with REST architectural design, so we can be able to connect Internet-enabled embedded devices and access them through universally accepted standards-based methods. In this subsection we highlight the CoAP and OMA LWM2M application protocols, since both protocols have been especially designed with the limitations of these devices in mind.

2.4.1 COAP

CoAP is a specialized Web transfer protocol optimized for resource constrained networks defined by the IETF CoRE Working Group [6]. CoAP is similar to HTTP but its goal is not to simply compress HTTP, but also implement a subset of REST operations optimized for M2M interactions. The interaction model is similar to the client/server model of the HTTP protocol. Clients request an action to a resource and then, the server sends the response with the status code. Messages are exchanged asynchronously over UDP.

CoAP has the following main features [6]:

- Use of the UDP binding to avoid costly TCP handshakes;
- Support for the methods defined in HTTP: GET, POST, PUT and DELETE. And also three types of response codes: 2.xx (success), 4.xx (client error), 5.xx (server error);
- URI based resource representation;
- Support for different payload content types;
- Support for Blockwise Transfers. It allows the transmission of larger amounts of data by splitting the data into blocks;
- A Resource Observe mechanism built using a publish/subscribe pattern;
- Resource discovery capabilities to allow clients to discover all resources handled by servers.

CoAP Messages: The CoAP frame format has a 4 byte fixed header and optional fields in Type-Length-Value (TLV) format as can be seen in Figure 2.8 [6].

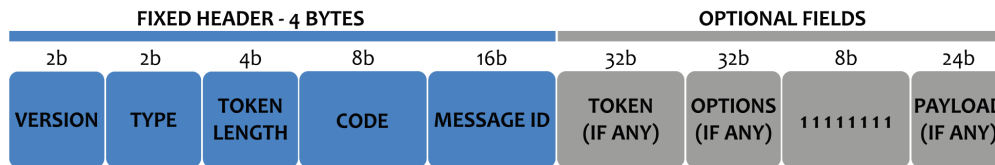


Figure 2.8: CoAP frame format.

To establish a message exchange between client and server, different message types can be used [6]:

- Confirmable (CON): this type of message is used when a reliable transmission is needed. Since messages are transported using UDP, this reliability is achieved with packet retransmission if a response is not received within a given timeout. However, the packet may be lost if the maximum number of retransmissions is reached;
- Non-Confirmable (NON): this type of message is used when a reliable transmission is not needed. It is quite useful for resources that are sent periodically;
- Acknowledge (ACK): this type of message is used as a response to acknowledge a CON request. It may carry response data (piggy-backed response) or not (separate response). The separate response is used when the server is not able to process the request immediately, but will process and send the response later;
- Reset (RST): this message indicates that a CON request has arrived but there is no context to process it.

An example of how CoAP works with request/response model and how it uses the type of messages and methods available are shown in Figure 2.9.

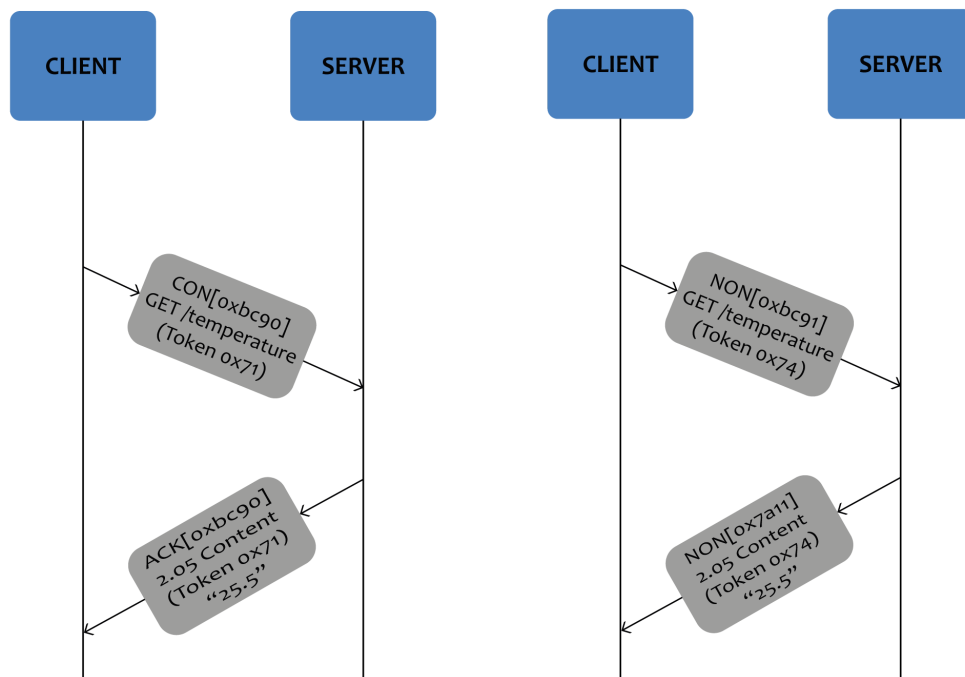


Figure 2.9: CoAP request/response model. Confirmable (left) and Non-confirmable (right). Adapted from [6].

In the rest of this subsection we will discuss some of the main CoAP features, such as: Blockwise Transfers, Resource Observe Pattern and Resource Discovery.

Blockwise Transfer: Since the maximum payload of 6LoWPAN packets is very constrained (about 81 bytes), applying even more network fragmentation might be even worse. So, in order to avoid operations that could cause fragmentation at the network level, with the use of this feature it is

possible to carry the data fragmentation from the network to the application layer [28]. This feature is particularly useful when a resource representation exceeds the number of bytes that can be transmitted in a single 6LoWPAN frame. Thus, one REST operation can be fragmented into multiple packets, without compromising the performance of the constrained 6LoWPAN network [29].

Resource Observe Pattern: In HTTP, the transactions are always initiated by the client. If a client wants to stay up-to-date about a specific resource status, it should perform GET operations again and again. In this type of networks, with limited resources, this polling model becomes very expensive. CoAP addresses this problem by providing an enhancement to the REST model: adding the observer pattern. Using this pattern, a client can indicate its interest in further updates from a resource by specifying the Observe option in a GET request. If the server accepts the option, the client becomes an observer of this resource and receives an asynchronous notification message each time it changes [30].

Resource Discovery: In typical CoAP applications the devices must be able not only to discover other devices on the network, but also the resources available on each one of them. This feature is already common on the Web, also called Web discovery in the HTTP community. The need for standardized way to perform resource discovery is much greater in constrained networks than on the current Web. As an answer, the IETF introduced a method for discovering and advertising resource descriptions, available through the /.well-known/core path (RFC5785) [31]. CoAP servers should include this method, so CoAP clients can be able to access all resource descriptions available on that server by simply performing a GET request on the /.well-know/core URI. The resource description retrieved by the server must be formatted according to the HTTP link header format [32].

2.4.2 OMA LIGHTWEIGHT M2M

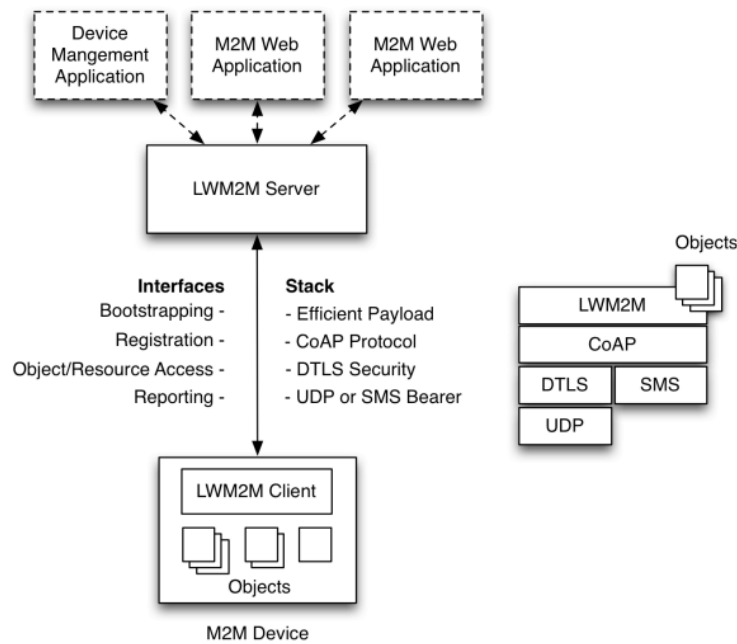


Figure 2.10: The Lightweight M2M architecture with the LWM2M Client and the LWM2M Server. Taken from [33].

The OMA Lightweight M2M (LWM2M) [34] was design with constrained devices in mind. This protocol provides a lightweight and compact secure communication interface along with an efficient data model, which together enable device management and service enablement for M2M devices. As with other device management standards, the Lightweight M2M solution is called an Enabler. The LWM2M Enabler defines the application layer communication protocol between a server and a client. The LWM2M Client typically resides on the embedded device and is integrated as a software library. The typical architecture diagram can be seen in Figure 2.10.

The LWM2M protocol has at least four distinguishing characteristics [33]:

- Features a modern architectural design based on REST, appealing to software developers;
- Defines a resource and data model that is extensible;
- Designed with performance and the constraints of M2M devices in mind;
- Uses for communication an efficient protocol already presented, the CoAP.

The LWM2M Enabler defines a simple Object/Instance/Resource model where each piece of information provided by the LWM2M Client is a resource. The resources are further organized into objects. The LWM2M Client can have several different resources, where each one of them belongs to an object [34]. The Objects/Resources are accessed with simple URIs: /Object ID/Object Instance/Resource ID.

The first release of the OMA LWM2M standard specifies, in addition to the Enabler itself, an initial set of objects for device management purposes [35]:

Object Name	Object ID	URN	Multiple Instances?	Description
LWM2M Security	0	urn:oma:lwm2m:oma:0	Yes	This LWM2M Object provides the keying material of a LWM2M Client appropriate to access a specified LWM2M Server.
LWM2M Server	1	urn:oma:lwm2m:oma:1	Yes	This LWM2M objects provides the data related to a LWM2M Server.
Access Control	2	urn:oma:lwm2m:oma:2	Yes	Access Control Object is used to check whether the LWM2M Server has access right for performing an operation.
Device	3	urn:oma:lwm2m:oma:3	No	This LWM2M Object provides a range of device related information which can be queried by the LWM2M Server, and a device reboot and factory reset function.
Connectivity Monitoring	4	urn:oma:lwm2m:oma:4	No	This LWM2M objects enables monitoring of parameters related to network connectivity.
Firmware	5	urn:oma:lwm2m:oma:5	No	This Object includes installing firmware package, updating firmware, and performing actions after updating firmware.
Location	6	urn:oma:lwm2m:oma:6	No	The GPS location of the device.
Connectivity Stats	7	urn:oma:lwm2m:oma:7	No	This LWM2M Object enables client to collect statistical information and enables the Server to retrieve these information, set the collection duration and reset the statistical parameters.

Figure 2.11: Standard Objects from LWM2M Technical Specification.

A typical message flow between a LWM2M Client and Server is shown in Figure 2.12.

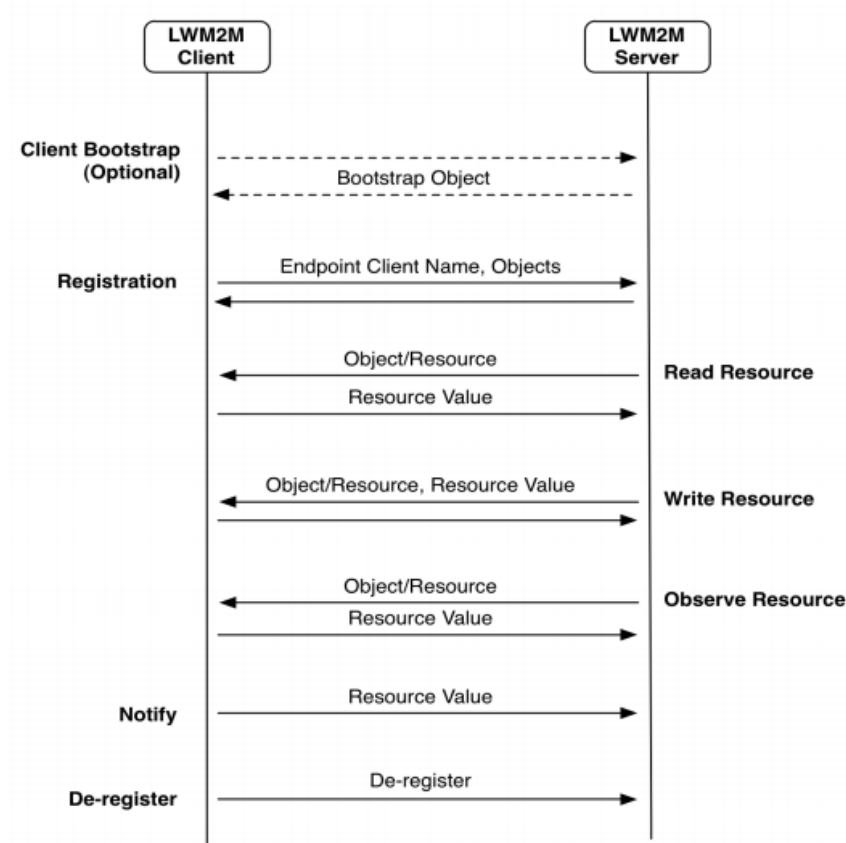


Figure 2.12: Abstract message flow example between a LWM2M Client and Server, the actual messages are mapped to CoAP requests and responses. Taken from [33]

One of the major benefits of LWM2M protocol is the abstraction that exists between the COAP protocol used for communication and the LWM2M data model. The creation of this standard data model enables interoperability between devices from different vendors. In addition, it is possible at any time to extend the data model, in order to support other type of functionalities for a particular application [33]. For example the IPSO Alliance has already created compatible object descriptions related to smart city applications. In the scope of this thesis, besides the set of objects for device management purposes presented before, we only implemented two more: the IPSO Temperature Sensor and the IPSO Light Control. In Figure 2.13 it's presented an overview of these two IPSO objects. All the information about LWM2M objects can be consulted in [35].

IPSO Temperature: This IPSO object should be used over a temperature sensor to report a remote temperature measurement. It also provides resources for minimum/maximum measured values and the minimum/maximum range that can be measured by the temperature sensor. The unit used here is Celsius degree.

IPSO Light Control: This IPSO object is used to control a light source, such as a LED or other light. It allows a light to be turned on or off and its dimmer setting to be control as a % between 0 and 100.

OBJECT INFO			
Object Name	Object ID	URN	Multiple Instances?
IPSO Temperature	3303	urn:oma:lwm2m:ext:3303	Yes
IPSO Light Control	3311	urn:oma:lwm2m:ext:3311	Yes

RESOURCE INFO - IPSO TEMPERATURE						
Resource Name	Resource ID	Access Type	Multiple Instances?	Type	Units	Description
Sensor Value	5700	R	No	Decimal	Celsius	This resource type returns the temperature value in Celsius degrees.
Min Measured Value	5601	R	No	Decimal	Celsius	This minimum value measured by the sensor since it is on.
Max Measured Value	5602	R	No	Decimal	Celsius	This maximum value measured by the sensor since it is on.

RESOURCE INFO - IPSO LIGHT CONTROL						
Resource Name	Resource ID	Access Type	Multiple Instances?	Type	Units	Description
On/Off	5850	R,W	No	Boolean		On/Off control.
Dimmer	5851	R,W	No	Integer	%	Proportional control, integer value between 0 and 100%.
On Time	5852	R,W	No	Integer		The time in seconds that the light has been on.

Figure 2.13: IPSO Temperature and Light Control Objects Overview. Adapted from [35].

OPERATING SYSTEMS FOR WIRELESS SENSOR NETWORKS

3.1 DESIGN ISSUES AND CHALLENGES

A Wireless Sensor Network operates at two levels: the network level and the node level. In the network level focus is on the connectivity, routing protocols, communication channel characteristics, etc. In the node level focus is on hardware, radio, CPU, sensors and limited power [7]. This section discusses the important issues and challenges to be considered while choosing an operating system for a street lighting mesh network. Such issues motivate the design requirements of an operating system needed for this specific WSN implementation. These design requirements are presented in section 3.2.

3.1.1 RESTRICTED RESOURCES

A typical sensor node for WSNs is constrained by limited battery power, processing capability, memory and bandwidth. The platform used in this thesis is not different. The 4LD is equipped with the PIC24FJ128GA308 processor, which has only 128Kb of flash program memory, 8kb of RAM and a maximum operating frequency of 32 MHz. This hardware platform is going to be introduced in more detail in Chapter 5.

Power: Power consumption is quite crucial to the life span of most WSN based applications, especially the ones that are battery powered. A typical node with a limited power supply has to operate for months to years [36]. In this specific implementation the power consumption is not a critical issue so far, mostly because the main application is street lighting control, therefore the sensor node is going to be connected to the mains electricity network. In other cases the sensor node and the street lighting system can be powered by solar panels. In this case, the power consumption becomes a very important issue and should be taken into account when designing an operating system for WSNs.

When compared to computation and sensing, the main source of power consumption in a WSN embedded device is the communication support [37]. For example, when transmitting one data bit over the RF is being spent energy that would be sufficient to execute thousands of instructions by the microprocessor. Besides the transceiver, readings and writings to the flash are also responsible for a significant energy consumption. So, when dealing with modular operating systems it is important to consider the energy consumption associated with the load/unload of modules into program memory [13].

The operating system must ensure the existence of efficient energy management and optimization mechanisms, in order to prolong the lifetime of the sensor nodes, especially of those that are battery operated. One of the mechanisms to conserve power is the periodic sleeping of the sensor nodes. Sensor nodes can typically operate in one of three sleep modes: idle, power down and power save. In the idle mode only the processor shuts off. In power down mode the sensor node shuts off everything except the watchdog timer and interrupt logic. The power save mode is similar to power down mode except that it keeps the timer running [37] [13].

Processing Power and Memory: Sensor nodes have a reduced processing power, usually in the order of a few MIPS [37]. In this specific case, a PIC24 microcontroller working only at 16 MIPS and operating at 32 MHz. Therefore, the computation of intensive operations must be properly scheduled. Otherwise, higher priority tasks can get delayed/starved [13].

Other main constraint for the developer is this available program memory. As already mentioned, the PIC24 only includes 128Kb of program memory. Hence, the operating system choosed for the WSN implementation should fit within this memory.

3.1.2 PORTABILITY

Portability is one of the key features required in an OS for WSNs. Taking into account the rapid evolution of the platforms these days, a software must be functional in a wide range of platforms. Thus, the OS should allow the port to different platforms without having to make major changes [13].

3.1.3 CUSTOMIZABILITY

As mentioned in Chapter 1, applications in WSN are spread across a wide range of disciplines (Figure 1.1). Different applications demand different requirements from the OS. These requirements may be reconfigurability and real-time guarantees, among others [13]. Hence, the OS design should allow for an easy customization and extension to different types of applications.

3.1.4 MULTITASKING

At a given point of time, a sensor node may have to perform more than one task/operation. Some of these are concurrent operations. Thus, if not handled carefully, higher priority tasks/operations may be delayed, beyond acceptable limits. The physical parallelism that is provided by the microcontrollers

is limited and the context switch overhead must not be neglected. Therefore, the OS must have a good execution model and an efficient mechanism to switch between different tasks [13].

3.1.5 NETWORK DYNAMICS

In terms of network dynamics in WSNs there are some aspects that must be taken into account, such as mobility, possible communication failures in channels/nodes, segmentation on the network, among others. Sensor nodes may be subjected to communication failures, which can be related to interference in the RF channel. That may lead the network to diverge from its normal behavior. Therefore, the operating system must provide mechanisms to facilitate the easy adaptation of the sensor node to the most diverse network conditions. This will provide transparency to the application from network dynamics [13].

3.1.6 DISTRIBUTED NATURE

In most cases, WSNs consist of thousands of nodes, usually spread over a wide geographic area. Thus, the operating system must handle this distributed environment, providing efficient management of the distributed nodes in order to make them look as a single virtual entity. This entails [13]:

- Inter-node communication and Failure Handling - Potential problems, such as low bandwidth, link failures and inaccessible nodes should be masked from the application point of view. Thus, the OS must be robust in order to handle these communication failures so that they do not interfere with the normal operation of the application;
- Heterogeneity - Heterogeneity is a feature presented in most of the WSNs deployments. They are usually composed by sensor nodes with different capabilities in terms of memory, processing power and may also have different sensing capabilities. Therefore, the OS must be capable of distributing the system load according to the capabilities of each node in order to mask this heterogeneity from the application user;
- Scalability - The OS must implement efficient algorithms, in order to avoid a large degradation of the network performance as the number of nodes increases.

3.2 DESIGN CHARACTERISTICS

After discussing the key issues to be considered while choosing an operating system for a street lighting network implementation, following the desired key OS characteristics are presented.

3.2.1 FLEXIBLE ARCHITECTURE

As already mentioned in section 2.2 the architecture of the kernel plays a major role in the way the OS provides services. The kernel architecture must ensure the possibility of adding new services

or modify existing ones at runtime - reconfiguration. It will also have an influence in the memory footprint of the core kernel [13].

Monolithic architectures form a single system image that agglutinates all the required OS services, resulting in a higher kernel size. This type of architecture doesn't offer flexibility when it comes to make changes to the kernel or to the application, since the entire system image must be replaced.

Modular architectures, in turn, have the ability to dynamically load/unload service modules. Because of that it is possible to bundle only the required services for an application, that are needed in a specific moment, in a single system image. This will slightly degrade the OS performance, since it introduces overheads when loading and unloading service modules. On the other hand, the kernel memory size is smaller when compared to monolithic architectures. Since this type of architecture allows to add services at run-time, they offer flexibility when updating or replacing the kernel without the need to replace the entire system image.

Virtual machine architectures also offer flexibility in the application development. As the application is constituted by specific instructions to the virtual machine, reconfiguration becomes easy [13].

In sum, we can say that the chosen architecture presupposes a compromise between performance and flexibility. The monolithic architecture is not ideal for applications that require regular updates. The modular and virtual machine architectures, in turn, are more suited if the application requirement is reconfiguration. It will simplify the code maintenance and modification problems. So, an OS for Wireless Sensor Networks should allow to easily add new services or updating the existent ones if required, maintaining the kernel memory footprint as small as possible.

3.2.2 EFFICIENT PROGRAMMING MODEL AND SCHEDULING

The programming model used by most of the embedded systems is event-based. However, the thread-based programming model can also be used. As already discussed, the thread-based models are usually easier to use, but less efficient, due to context switches overhead, memory consumption, etc [14]. On the other hand, event-based models are usually more efficient, but very difficult to use in large designs [15]. The comparison between the two can be found in Table 2.1.

The programming model must define efficient synchronization mechanisms during the access to shared resources or information, so as to avoid race conditions. It should also provide an efficient scheduler, that performs concurrent intensive tasks and prevents tasks from being blocked from execution [13].

The scheduling of computational units must also be performed efficiently, especially in critical applications. In these applications, the tasks must be executed within certain time limits, otherwise it may lead to catastrophic situations. In our specific application temporal requirements are not hard, because the result of the task execution retains some utility to the application, even after a temporal limit, although it will lead to a degradation of Quality of Service (QoS). In subsection 3.2.6 this issue is discussed in more detail.

3.2.3 CLEAR APPLICATION PROGRAMMING INTERFACE

APIs are responsible for providing a layer of abstraction between the low-level functionalities and the application. The OS must have a broad range of APIs to interact with the system and its I/Os.

This will be crucial for the application developer, since he/she will not need to consider the low level functionalities of the sensor node hardware [13]. The chosen OS may include various APIs, such as:

- Networking API;
- Sensor data reading API;
- Memory management API;
- Power management API;
- Task management API (Set delays, set priorities, post events).

The APIs that are responsible for memory management are important if the developer wants to reconfigure the software running on the node dynamically. APIs related to posting of events and setting the delays of tasks gives greater flexibility when scheduling them. With these and other APIs the developer can then build applications and use the available resources efficiently.

3.2.4 REPROGRAMMING

Reprogramming is a crucial feature that the chosen OS must have, since it will simplify the software management in the sensor nodes. This feature, which allows dynamic updating of the sensor nodes software, is crucial in WSNs because the sensors are often deployed in locations that are difficult to access and also due to the fact that these networks usually are composed by hundreds or even thousands of nodes [13]. In the target street lighting network, reprogramming is also an important characteristic. Without reprogramming it is difficult to add, modify or delete software from the running system, not to mention that manual reprogramming not only entails high monetary costs, but also takes a lot of time to update/upgrade the full network, especially if the network has a high number of nodes.

To avoid all of these costs, the code should be distributed over the air using code dissemination protocols [38] [39]. These protocols will perform the splitting and compression of the code to be sent to update the nodes. For this operation be successfully carried out the code should be relocatable, which means that the code should run in any location of the memory. Therefore, the operating system must provide mechanisms for dynamic allocation of memory in order to facilitate loading/unloading of software components at run-time.

Reprogramming can be performed with different granularities, which can range from the setting of a variable to changing the entire software image of the node. Application level reprogramming will replace the entire application image of the node. Modular level reprogramming only replaces a module for an application. Instruction level and variable level reprogramming gives more flexibility if tuning parameters of the application is needed. Figure 3.1 shows the different levels of granularity that each OS is able to support in reprogramming.

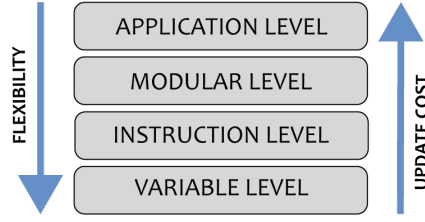


Figure 3.1: The reprogramming flexibility and update cost, depending on the level of granularity. Adapted from [13].

3.2.5 RESOURCE MANAGEMENT

The OS of election should perform the management of its resources (processor, memory, battery, etc.) efficiently. An efficient use of the processor can be achieved by using a scheduler with an optimal scheduling policy. Regarding the efficient memory use, it requires the existence of memory protection mechanisms, dynamic memory allocation, etc. The battery should also be treated as a special resource, therefore the use of the sleep modes becomes imperative in order to avoid unnecessary energy wastes [13]. The power management interfaces that are provided by an OS should impose an optimized management of the available energy, in order to prolong the life span of the embedded devices.

3.2.6 REAL TIME NATURE

This design characteristic may or may not be required, depending on the specific application. Real-time tasks can be classified into periodic and aperiodic. A task that is responsible for monitoring temperatures is an example of a periodic task. Smoke detection is an example for an aperiodic task. These tasks can also be classified into critical and non-critical. A critical task requires that its execution must occur in a stipulated time, otherwise it may lead to catastrophic consequences [13].

As stated before, in our specific application temporal requirements are not critical, although in order to improve the quality of the network communication and to offer Quality of Service (QoS) guarantees to the users, the round trip time (RTT) should be analyzed. The RTT is the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgment of that signal to be received. In our case, even a RTT in order of 1s is feasible, but since we want to provide a good QoS, this metric should be optimized as much as possible.

3.3 EXISTING OPERATING SYSTEMS

In recent years, we have seen the emergence of multiple operating systems specially designed for applications in WSNs. Those who had more prominence within the community are: TinyOS, Contiki, Nano RK, MANTIS and LiteOS. These operating systems will be strictly analyzed in the next subsections taking into account the design characteristics identified in section 3.2.

3.3.1 TINY OS

TinyOS [40], developed at UC Berkeley, was one of the first operating systems that have emerged for WSNs. TinyOS can support concurrent programs with very low memory requirements. The OS has a footprint that fits in 400 bytes [41].

Architecture: TinyOS (TOS) has a monolithic architecture. It is composed by components that may have three possible computational abstractions: commands, events and tasks. Commands and events are used to provide inter-component communication. Tasks, in turn, are used to provide intra-component concurrency. An event indicates the completion of a service, while a command signals a request to perform a particular service. Since this OS provides a single shared stack, there is no separation between kernel space and user space [41].

Programming Model: Older versions of this OS did not provide support for multithreading, the programming model was strictly event-driven. However, version 2.1 already supports multithreading. A high priority kernel thread is allocated to the scheduler. Message passing is used for communication between the applications threads and the kernel threads. The applications make system calls by posting tasks to the kernel thread. The kernel, in turn, will preempt the thread that is currently running and will execute the system call. Using this mechanism it is guaranteed that only the kernel executes TinyOS code [41], however, it introduces an overhead of 0.92% [40].

Scheduling: Regarding scheduling, TinyOS uses the FIFO technique. The scheduler does not allow running tasks to be preempted by other tasks, however, a running task can be preempted by interrupt handlers, commands or events. One of the main disadvantages of the FIFO scheduling used is that it can be unfair for short tasks that are waiting behind time-consuming tasks [41].

Resource Management: The sensor nodes hardware don't offer memory protection mechanisms, so it is the responsibility of the OS to provide such mechanisms. The TinyOS uses a static memory management approach and the version 2.1 also incorporates memory safety [41]. Since the embedded devices are resource-constrained, the use of low level languages like nesC is strictly necessary [42]. In terms of energy management, the TinyOS provides an API in order to conserve and manage power properly. This API enables the processor to sleep whenever possible upon the next clock after the following conditions are met: the radio is off, all clocks interrupts are disabled and task queue is empty [13].

Communication Protocol Support: This OS implements two multi-hop protocols [41]:

- Dissemination - Reliably data transfers to every node in the network [43]. It enables reconfiguration and reprogramming;
- TYMO - Implementation of the DYMO protocol, a routing protocol for mobile ad hoc networks. TYMO protocol is implemented on the top of the active messaging stack [44].

TinyOS supports application level reprogramming. If the developer wants to make updates to the TinyOS application, he must do it by modifying the source code directly. Then, he must recompile the TinyOS and place the new image on the sensor node [39]. As the entire image has to be flashed, reprogramming causes a high communication overhead. This is due to the monolithic architecture of this OS [13]. Recent versions of this OS (version 2.1.1) also incorporates a 6LoWPAN/IPv6 stack.

3.3.2 CONTIKI OS

Contiki is a lightweight, portable and open-source OS written in C, specially tailored for WSN applications. Contiki's memory footprint is about 2 kB of RAM and 40 kB of ROM [45].

Architecture: This OS follows a modular architecture, while the programming model is event-driven. Contiki comprises an event scheduler, which is responsible for dispatching events to running processes. An event handler, in most cases, will run until completion, however, these handlers can use internal mechanisms to allow preemption. In Contiki there are two types of events: asynchronous and synchronous. Synchronous events resemble a call of a function. They are dispatched immediately, being delivered directly to the target process. In turn, asynchronous events are held on an event queue and dispatched later. The kernel will loop through this queue and will deliver the event to the target process or processes. Process execution as well as being triggered by events, can also be triggered by polling mechanisms. The Contiki's polling mechanisms can be seen as special type of events that are scheduled between each asynchronous event. If a poll is scheduled, all processes that implement a poll handler are called, according to their priority [41]. These mechanism is used to invoke a process from an interrupt context. Figure 3.2 shows an overview of the Contiki OS architecture.

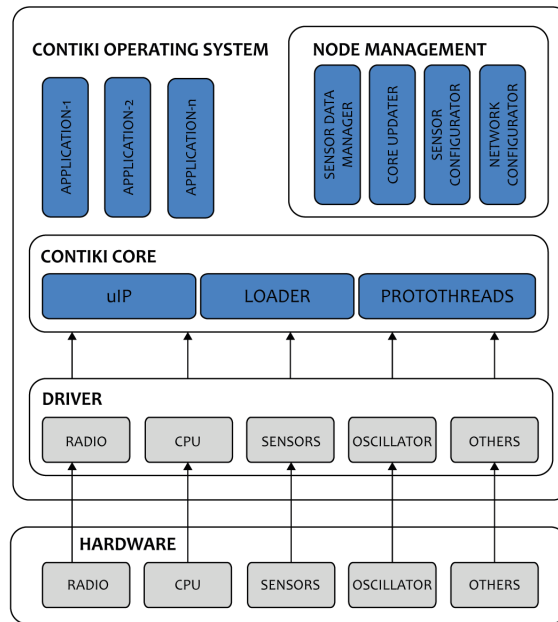


Figure 3.2: Contiki architecture overview. Adapted from [46].

Programming Model: Contiki has a hybrid kernel, hence follows the event-driven model, but at the same time supports multi-threading. In Contiki multi-threading is supported using a library that runs on top of the event-driven kernel. Whenever an application needs multi-threading, this library can be linked. This library has two essential parts: one part is responsible for interaction with the kernel - platform independent part - and the other part is responsible for implementing mechanisms of stack switching and preemption - platform specific part [41]. Contiki uses protothreads to implement the multithreading library [47]. The Contiki protothreads were specially tailored for application in

resource-constrained devices. They allow to wait for any incoming events without blocking the whole system. They don't require a separate stack for each thread, thus the overhead and the large memory consumption introduced when allocating multiple stacks is avoided. In sum, protothreads have the following benefits:

- Very lightweight;
- A very small overhead of only two bytes per protothread;
- No need for a separate stack;
- Highly portable, since they are written in C.

Scheduling: As mentioned earlier, Contiki comprises an event scheduler, that dispatchs the events to the respective processes. Usually they run to completion, but can be preempted by interrupts [41].

Resource Management: The memory management in Contiki is performed in dynamic form, which allows also the dynamic linking of programs [41]. Contiki comprises several modules for allocate/deallocate memory. The most important one is the Managed Memory Allocator as it avoids memory fragmentation problems [48].

Communication Protocol Support: Contiki was the first operating system that opened up the possibility of using wireless sensor nodes along with IP communications. This was achieved through the uIP, a TCP/IP protocol stack. In 2008, this OS incorporated the uIPv6, the world's smallest IPv6 stack [41]. In terms of memory size, the footprints of these two stacks are very small. The uIP stack consumes less than 5kB and the uIPv6 stack approximately 11kB. This makes them ideal for use in constrained environments. Besides uIP and uIPv6 stacks, Contiki also provides another protocol stack for network-based communication, called Rime. Rime protocol offers single hop unicast, single hop broadcast, and multi-hop communication support [41]. Contiki also has an implementation of the RPL protocol already introduced in Section 2, called ContikiRPL [49].

3.3.3 LITE OS

LiteOS [50] is a Unix-based OS also tailored for WSNs applications. It was developed at the University of Illinois. LiteOS has a very small memory footprint, which makes possible its operation in sensors with only 8 MHz of CPU frequency, 128 bytes of program flash, and 4 kB of RAM.

Architecture: This OS has a modular architecture and is divided into three subsystems [41]:

- LiteShell - LiteShell is very similar to the well-known Unix shell. It also allows file and process management using shell commands. As this shell resides on a PC it is possible the implementation of more complex commands, since it has more available resources. Some of the processing is done in this local PC, like the parse of the user commands by the shell, and then it is transmitted over the air to the target node;
- LiteFS - It is the LiteOS file system. In this OS, a WSN can be seen as a directory, which has within a file with a list of all neighboring sensor nodes;
- Kernel - Offers multi-threading support and dynamic loading of components. It also provides registration of event handlers through callback functions and synchronization mechanisms.

In Figure 3.3 the LiteOS architecture is presented.

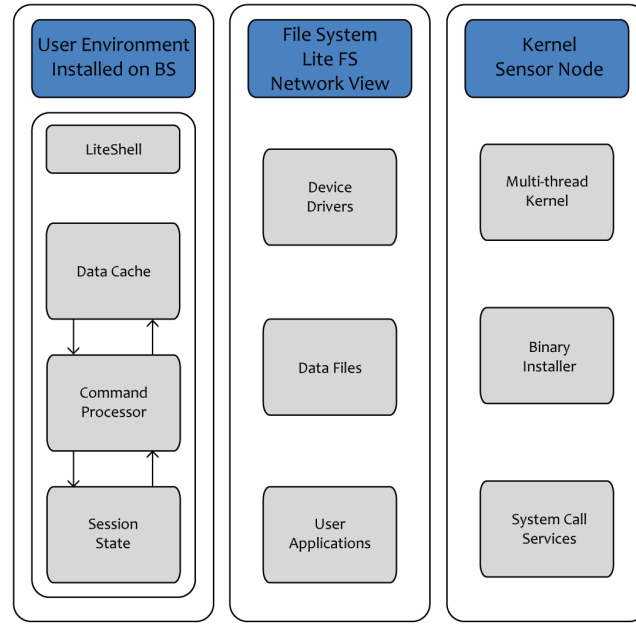


Figure 3.3: LiteOS architecture. Adapted from [41].

Programming Model: As previously stated, LiteOS supports multi-threading. In order to avoid potential errors that could occur when using shared memory space, each LiteOS thread has its own memory space. Besides support for multi-threading, LiteOS also offers event handling mechanisms [41].

Scheduling: Regarding scheduling policies, LiteOS implements round robin and priority-based algorithms. When a task is ready to be executed is added to the ready queue. Among all the tasks listed in this queue, the scheduler using a priority-based scheduling, will choose the next one to execute. Tasks only run to completion if all resources needed are available in the moment. Otherwise, it will enable the interrupts and enter in sleep mode. The task will resume its execution from where it had left, as soon as the required resources become available. As happens with other OSs mentioned above, it is possible that a high-priority task may have to wait behind a low priority one. This happens if the high-priority task enters the ready queue when the low priority one is running. In this case the low-priority task will run to completion, while the high-priority may miss its deadline. Therefore, the LiteOS should not be used for real-time applications [41].

Resource Management: In terms of memory management, LiteOS implements dynamic memory allocation functions (C-like malloc and free functions). The applications can use these functions to allocate or de-allocate memory at run-time. The dynamic memory will grow in the opposite direction of the LiteOS stack [41].

Communication Protocol Support: This OS provides support for communication through files. For each device on the sensor node LiteOS will create a file. For example, the radio interface will have a specific file. Each time data needs to be sent over the radio interface, the data is placed in this file, and only then transmitted [50].

3.3.4 NANO-RK

Nano-RK [51] is a preemptive multitasking OS, that can be used for hard and soft real-time applications. It has a very small memory footprint: consumes 2 Kb of RAM and 18 Kb of ROM.

Architecture: Nano-RK has a monolithic architecture [51]. The main applications of this OS are critical applications in which tasks must necessarily meet its deadlines. In order to be able to achieve this, the priority of each task, the deadlines and periods must be defined statically offline. In this way it is possible to ensure before execution if the deadlines are being met or not. It also has APIs providing dynamic configuration at runtime of the different task parameters - priority, period, deadline. However, the use of these APIs is not recommended since it is not possible to know *a-priori* if the deadlines will be met [41].

Programming Model: Nano-RK is a preemptive multitasking OS. The scheduler before selecting a task to be executed needs to safeguard the current status of the running task. And as discussed earlier, these context switches may lead to a memory consumption and energy increase, as well as a performance decrease. In this OS each task has a corresponding Thread Control Block (TCB). The TCB stores different information, such as [41]:

- Register contents;
- Task priority;
- Task period;
- Reservation sizes;
- Port identifiers.

Based on the task period, this OS has two linked lists of TCB pointers, so it can be able to order the set of active and suspended tasks. In order to maintain the correct state of shared data/resources, Nano-RK also offers synchronization primitives: mutexes and semaphores.

Scheduling: Since Nano-RK uses a preemptive priority driven scheduling algorithm OS, the CPU is always allocated to the highest priority task. In this case, if a low priority task is running when a higher one appears, the low priority one will be preempted and the CPU will be allocated to the highest priority one. For real-time periodic tasks, Nano-RK uses a rate monotonic scheduling algorithm. It also implements a rate harmonized scheduling algorithm for energy saving [52]. This algorithm attempts to group the execution of different tasks, so the CPU idle cycles can be eliminated [41].

Resource Management: Regarding memory management, Nano-RK uses a static approach. In Nano-RK, both the OS and applications reside in a single address space [41].

Communication Protocol Support: Regarding communication, Nano-RK has a protocol stack that provides an abstraction very similar to sockets. Whenever an application needs to send data, to initiate the communication it must create a socket. An application can also bind and listen to a particular port number to receive data [41].

3.3.5 MANTIS

The Multimodal system for NeTworks of In-situ wireless Sensors (MANTIS) is a multithreaded, lightweight and energy efficient operating system specially designed for WSNs [53]. The memory footprint, including kernel, scheduler and network stack, is only 500 bytes.

Architecture: As shown in Figure 3.4, the MANTIS owns an architecture organized in layers. Each of these layers is responsible for providing a range of services. In Figure 3.4 the services implemented in each layer are presented.

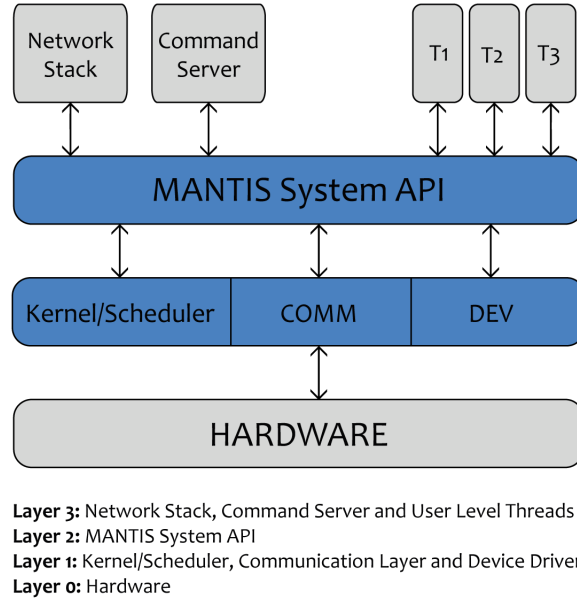


Figure 3.4: MANTIS architecture. Adapted from [41].

The only responsibility of the MANTIS kernel is dealing with the timer interrupts. All remaining interrupts are directly forwarded to the corresponding device driver. This device driver will post a semaphore, whenever it receives an interrupt. The semaphore, in its turn, will activate a waiting thread, that is responsible for handling the interrupt [41].

Programming Model: The MANTIS programming model is thread-based. MANTIS has a part of the RAM that works as a heap. When a thread is created, stack space is allocated from this heap. This stack space is freed when the thread exits. The MANTIS kernel is responsible to manage the thread table, which contains one entry for each thread. Since the MANTIS statically allocates memory for this table, there can only be a fixed maximum of threads. By default it is 12, although this maximum can be adjusted at compilation. The thread table contains different data, such as [41]:

- Current stack pointer;
- Stack boundary information;
- Pointer to thread function;
- Thread priority;
- Pointer to next thread.

This OS uses semaphores and binary mutexes, in order to avoid race conditions.

Scheduling: MANTIS makes use of priority-based scheduling algorithms, also allowing preemption. In this OS there are several priority classes: Kernel, Sleep, High, Normal and Idle. Within each of these classes round-robin scheduling algorithms are used. The CPU will always execute the highest priority thread in the ready queue. To each thread is assigned one time slice to run of 10ms. After that the CPU will preempt the running thread and will execute another thread for more 10ms. This context switch is done using timers. A system call or a semaphore post can also be able to trigger a context switch. If there are no threads ready to execute, the system will enter sleep mode in order to save energy. As already stated, the scheduler uses round robin scheduling, which may mean that higher priority tasks can cause the lower priority tasks to be ignored. This OS may also be able to accomodate real-time tasks, but it still needs to support a real-time scheduler, like the Rate Monotonic or the Earliest Deadline First [41].

Resource Management: In terms of memory management, MANTIS uses a dynamic approach. However, its use is not recommended since it could introduce a significant overhead [41].

Communication Protocol Support: This OS has the network stack divided in two main parts:

- Layer 3 (and above) protocols - implemented in user space, so it offers flexibility. However, the performance is reduced, since the network stack has to use the APIs provided by MANTIS instead of communicating directly with the hardware. This will lead to many context switches, introducing computational and memory overheads;
- COMM layer - implements synchronization, MAC and PHY layer mechanisms, also providing a clear interface to communicate with device drivers.

3.4 EVALUATION OF THE OPERATING SYSTEMS

After looking at the available operating systems for WSNs in the literature and taking into account the main issues and requirements of our application, as presented in section 3.1, we compare and evaluate the supported features in Figure 3.5.

OS	Architecture	Programming Model	Memory Management	Communication Protocol	Programming Language	Footprint (Program Memory)	Real Time Nature	Reprogramming	Open Source
TinyOS	Monolithic	Threads and Events	Static	Active Message	nesC	400 bytes	✗	✓	✓
Contiki	Modular	Protothreads and Events	Dynamic	uIP, uIPv6 and Rime	C	40 Kbytes	✓	✓	✓
MANTIS	Layered	Threads	Dynamic	COMM Layer	C	500 bytes	✓	✓	✓
Nano-RK	Monolithic	Threads	Static	Socket	C	18 Kbytes	✓	✗	✓
LiteOS	Modular	Threads and Events	Dynamic	File based communication	LiteC++	128 bytes	✗	✓	✓

Figure 3.5: Operating Systems Summary.

After analysing the documented features the OSs that have recently attracted more attention are TinyOS and Contiki OS. Since the main goal of this thesis is to implement a Wireless Mesh Network using IPv6, the only OS that encompasses a certified IPv6 stack is the Contiki OS. In fact, the Contiki operating system is the one that aroused more interest in the industry field and is Europe's leading operating system for sensor networks. The Contiki's IPv6 stack was contributed by Cisco and was, at the time of its release, the smallest IPv6 stack to receive the IPv6 Ready certification. Contiki also has a full implementation of IPv6 with TCP, UDP, RPL and ICMP. It supports IPv6 over IEEE 802.15.4 by providing a 6LoWPAN adaptation layer, a variety of duty cycled MAC layers and radio drivers for a variety of sensor motes and hardware platforms.

As stated before, Contiki also incorporates energy saving features, a lightweight multi-threading library (the protothreads), a straight forward programming style in C and a rich API library. One of these APIs measures the time spent running the various sensor node components (energy profiling library). This library eases the development of low power applications, since it gives accurate insights about where the energy is spent when an application is running. In addition to all the benefits highlighted above, also provides efficient mechanisms of energy management and has a very small memory footprint. Therefore, Contiki is the perfect operating system to implement the street lighting mesh network desired. In the next chapter all the features and functionalities offered by this OS will be presented and discussed.

THE CONTIKI OPERATING SYSTEM

4.1 BRIEF INTRODUCTION

Nowadays, of all the microprocessors that are sold, only 2% are used in PCs, while the remaining 98% are used in embedded systems [54]. Unlike PC computers that have plenty of resources, the embedded devices are very resource-constrained (memory, processing power, etc). Moore's Law predicts that in the future these devices will become increasingly smaller and cheaper. This means that it will be possible to deploy sensor networks in large areas with competitive prices. However, it does not imply that the resources will be less constrained [45]. So, the chosen operating system, the Contiki, must perform an efficient management of the available resources.

The operating system detailed in this Chapter is tailored to be used in embedded systems based on the MSP430, AVR, ARM, x86 and other architectures. This OS aims for maximum portability and, therefore, it is written in C. It is a feature-rich operating system. However, only some of its features are described and actually used on our implementation.

Contiki is developed by a group of talented engineers from industry and academia, lead by Adam Dunkels from the Swedish Institute of Computer Science [48]. The actual development can be followed in an online repository that can be accessed through the Contiki homepage at <http://www.contiki-os.org/>.

4.2 MAIN FEATURES

The Contiki OS comprises several features that are used on hundreds of embedded devices in diverse industry applications: car engines, oil boring equipments, satellites, container security systems,

among others [54]. It is also used in academic research projects and in university project courses all over the world. In this section we highlight the more interesting features and, also, those that were crucial to the development of this thesis, namely:

- Protothreads - Lightweight stackless threads in C [45]. Protothreads are discussed in more detail in Section 4.4;
- uIP stack - TCP/IP communication stack, that allows Contiki to communicate over both IPv4 and IPv6 [48]. Before Contiki's uIP stack, the embedded world considered IP to be too heavyweight. The IP implementations used in general computers could not fit in the constrained memories of the embedded devices [7]. The uIP stack is explained in more detail in Section 4.10;
- Rime stack - Layered communication stack that uses much thinner layers than traditional architectures [55]. Provides a set of communication primitives such as: single-hop unicast, single-hop broadcast and multi-hop [48], [55]. The Rime stack is presented with more detail in Section 4.10;
- Coffee File system (CFS) - Small and easy to use filesystem. The CFS is explained in Section 4.7;
- Contiki Toolkit (CTK) - A graphical user interface;
- Executable Linkable Format (ELF) Loader - Loads object files into a running Contiki system. This loader will be introduced in more detail in Section 4.8.

4.3 KERNEL AND PROCESSES

The Contiki is an event-driven kernel, but also supports preemptive multi-threading. The multi-threading feature is achieved through a library that could be linked with programs that require multi-threading [45]. The kernel doesn't contain any platform specific code. It implements only CPU multiplexing and lets the device drivers and the applications communicate directly with the sensor node hardware [45]. Each application in Contiki is implemented and run as a process. To implement a process in Contiki, protothreads are used. Both the kernel and applications use the protothreads extensively, in order to achieve cooperative multitasking. Every Contiki process is composed by:

- Process control block (PCB) - has run-time information about the process, such as: a textual name of the process, a pointer to the process thread and the information about the process state. The PCB is not defined directly, but through the *PROCESS()* macro [56]. The process control block is shown in Listing 1. Each field of the process control block is described in Table 4.1;
- Process thread - contains the code of the process and is implemented as a single protothread, that is invoked from the Contiki process scheduler [56]. An example process thread is shown in Listing 2.

```

struct process {
    struct process *next;
    const char *name;
    int (* thread)(struct pt *, process_event_t, process_data_t);
    struct pt pt;
    unsigned char state, needspoll;
};

```

Listing 1: Process control block in Contiki OS. Taken from [56].

```

PROCESS_THREAD(hello_world_process, ev, data)
{
    PROCESS_BEGIN();

    printf("Hello, \uworld\n");

    PROCESS_END();
}

```

Listing 2: Process thread example in Contiki OS. Taken from [56].

Field	Description
<i>next</i>	Points to the next PCB in the linked list of active processes
<i>name</i>	Points to the textual name of the process
<i>thread</i>	Points to the process thread
<i>pt</i>	Holds the state of the protothread in the process thread
<i>state</i>	Internal flag that keep the state of the process
<i>needspoll</i>	Set by the <i>process_poll()</i> function when the process is polled

Table 4.1: Process control block fields description.

In Contiki, the code can run in either two execution contexts:

- Cooperative - Code never preempts other code. The processes always run in this execution context [56];
- Preemptive - Can preempt the execution of cooperative code. Only interrupt service routines and real-time timers run in this execution context [56].

The code running in both execution contexts is presented in Figure 4.1.

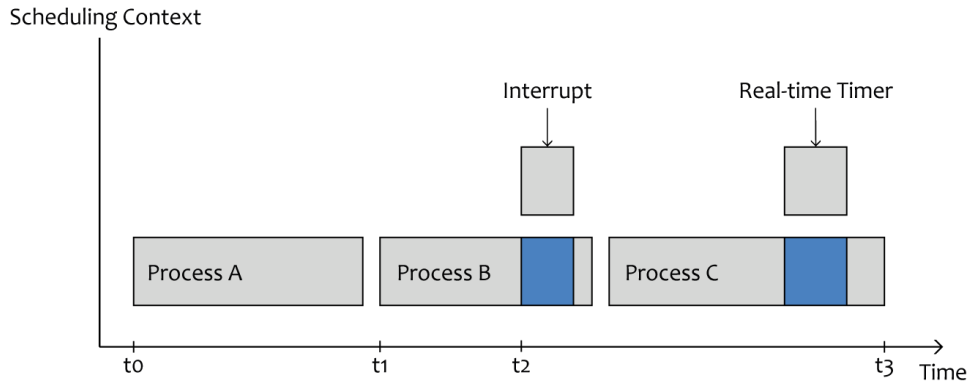


Figure 4.1: Contiki execution contexts. Adapted from [56].

4.3.1 EVENTS

The communication between processes in Contiki is achieved by posting events [45]. There are two types of events in this OS: synchronous and asynchronous events.

Synchronous events: This type of events is directly delivered to the target process, and can only be posted to a specific process [56]. Since they are delivered immediately, posting this type of events is similar to a function call: the process to which the event is delivered is directly invoked, and the process that has posted the event is blocked until the receiving process has finished processing the event [56].

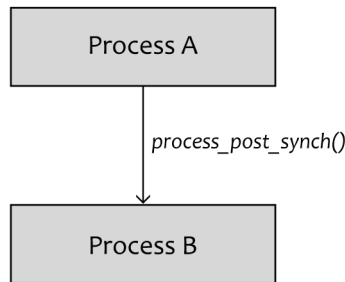


Figure 4.2: Synchronous event.

Asynchronous events: Unlike synchronous events, the asynchronous events are not directly delivered to the receiving process, but only some time after they have been posted. Before the event has been delivered, this type of events are held on an event queue inside the kernel, as can be seen in Figure 4.3. The kernel will loop through this event queue and then will deliver the event. The receiver of an asynchronous event can be a specific process or all running processes. If the event receiver is a specific process: the kernel will deliver the event by invoking this process. But if the event receiver is set to be all the running processes: the kernel sequentially delivers the same event to all the processes [56].

The asynchronous events are posted through the *process_post()* function. This function will check first if there is space for more events in the queue, by checking the current event queue size. If there is space, it will insert the event at the end of the event queue. If not, the function will return an error [56].

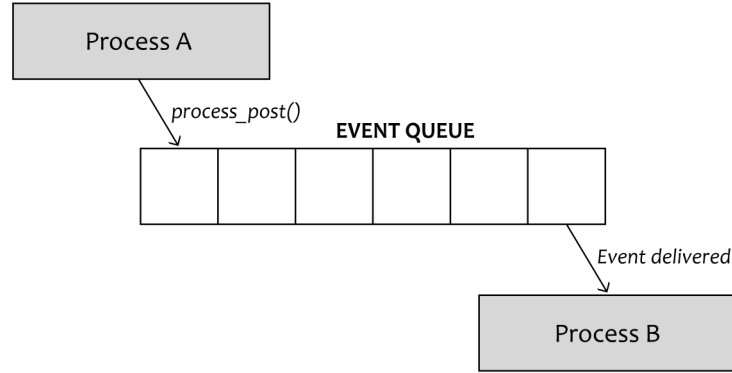


Figure 4.3: Asynchronous event.

4.3.2 PROCESS POLLING

The process poll requests are special type of events. A process can be polled by calling the *process_poll()* function. When this function is called on a specific process, this process will be scheduled as soon as possible. Process polling is useful to make a process run from an interrupt handler. The *process_poll()* is the only function that is safe to call from the preemptive execution context [56].

4.3.3 THE PROCESS SCHEDULER

The process scheduler is responsible for invoking processes when it's their time to execute. This is done by calling the function that implements the process thread. The scheduler will invoke processes either in response to an event that has been posted to a process, or a poll that has been requested for the process. When invoking a process in response to an event that has been posted, the scheduler passes the event identifier to the invoked process and also will pass an opaque pointer, that may be set to NULL to indicate that no data is to be passed. On the other hand, when a poll is requested for a process, no data can be passed [56].

The Contiki scheduler will make sure that the processes that have the field *needspoll* active, will have higher priority, and in the case that there are more than one process with this field active, these processes will be dispatched sequentially. By observing the variable *poll_requested*, the scheduler will know about the existence of processes that have the field *needspoll* active. If this variable is active, the scheduler will run the *do_poll* command to schedule the process that needs to be polled. Otherwise, if the variable *poll_requested* is NULL, there are no processes to be polled, so the scheduler will schedule the next event in the asynchronous event list [56]. In Figure 4.4 an general overview of the Contiki scheduler operation is presented.

Starting Processes: Start a process in Contiki is done through the *process_start()* function. This function is responsible for setting up the process control structure, place the process on the list of active processes and, also, call the initialization code in the process thread [56].

Autostarting Processes: In Contiki it is also possible to start the processes automatically when the system is booted, or when a specific module is loaded. This is done through the autostart module. When a specific module is loaded into Contiki, the autostart module will be responsible to inform the system about all the active processes the module contains [56].

Exiting and Killing Processes: Contiki processes can exit in one of two ways: either the process itself exits or it is killed by another process. A process can exit itself by calling the *PROCESS_EXIT()* function or when its execution reaches a *PROCESS_END()* statement. A process can kill another process through the *process_exit()* function. Whenever a process exits, the kernel will inform all the running processes of the process exiting (sends an event), so the other processes can free up any resource allocations made by this process. Then, the process will be removed from the list of active processes [56].

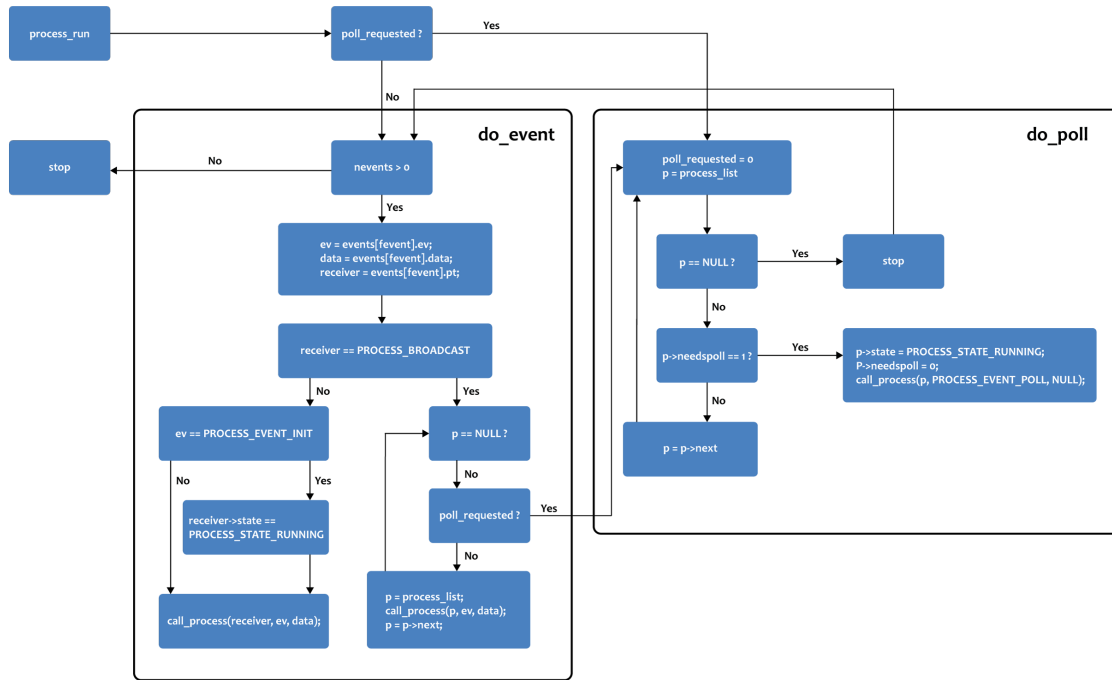


Figure 4.4: Process scheduling and process polling in Contiki. Adapted from [57].

4.4 PROTOTHREADS

The Contiki prototreads allow to wait for any incoming events without blocking the whole system. This solution is achieved using a C *switch* statement and a variable containing the position where the function was blocked [58]. The function will continue from this position when it is later invoked and will run until blocking or exiting.

The greatest benefit of protothreads over ordinary threads is, since it doesn't require a separate stack, the overhead and the large memory consumption introduced when allocating multiple stacks is avoided [58]. Each protothread only needs a few bytes for storing the execution state.

The protothreads API has four main operations [54] [58]:

- *PT_INIT()* - Initializes the protothread;
- *PT_BEGIN()* - Begins the protothread execution;
- *PT_WAIT_UNTIL()* - Conditional blocks the protothread;
- *PT_YIELD()* - Unconditional blocks the protothread;
- *PT_END()* - Exits the protothread.

Since the protothreads are implemented in C, the library can be used everywhere, where the C toolchain is available. However there are some constraints associated [56] [58]:

- Since they are stackless, a protothread can only run within a single C function;
- They don't offer a way to store local variables. We need to prepend them with the *static* keyword. Using this keyword these variables will be stored into the data segment;
- Since they are implemented using a C *switch* statement that cannot be nested, the protothreads can not use *switch* statements.

4.4.1 PROTOTHREADS IN PROCESSES

Each one of the Contiki's processes implements its own version of protothreads. However, the statements used in processes are slightly different than the pure protothread statements presented in the section 4.4. Process-specific protothread macros that are used in Contiki processes are shown in Table 4.2.

Macro	Description
<i>PROCESS_BEGIN()</i>	Declares the beginning of a process' protothread
<i>PROCESS_END()</i>	Declares the end of a process' protothread
<i>PROCESS_EXIT()</i>	Exit the process
<i>PROCESS_WAIT_EVENT()</i>	Wait for any event.
<i>PROCESS_WAIT_EVENT_UNTIL()</i>	Wait for an event, but with a condition
<i>PROCESS_YIELD()</i>	Wait for any event, equivalent to <i>PROCESS_WAIT_EVENT()</i>
<i>PROCESS_WAIT_UNTIL()</i>	Wait for a given condition. May not yield the process
<i>PROCESS_PAUSE()</i>	Temporarily yield the process

Table 4.2: Process-specific protothread macros. Adapted from [56].

4.5 PREEMPTIVE MULTI-THREADING

As already stated before, Contiki is an event-driven OS, but also supports preemptive threads. The preemptive threads are implemented through the mt library [45]. This library offers an alternative method for concurrent programming in Contiki. The mt threads have private stack segments and

program counters that are stored when a context switch happens. Unlike protothreads, the mt library allows preemptive scheduling with the possibility to yield from nested functions in a thread [59]. A mt thread can have three different states throughout its lifetime [59]:

1. *MT_STATE_READY* - Once a thread is first started by applying *mt_start()* on it, the mt library sets its state to *MT_STATE_READY*. This state specifies that the thread is ready to execute. The thread can be executed by calling the *mt_exec()* function. When the thread is yielded by calling the *mt_yield()* function, the ready state is reseted;
2. *MT_STATE_RUNNING* - is the state of a thread that is currently being executed;
3. *MT_STATE_EXITED* - is the final state of a thread, specifying that it can not be executed anymore.

The Figure 4.5 shows the state chart of threads.

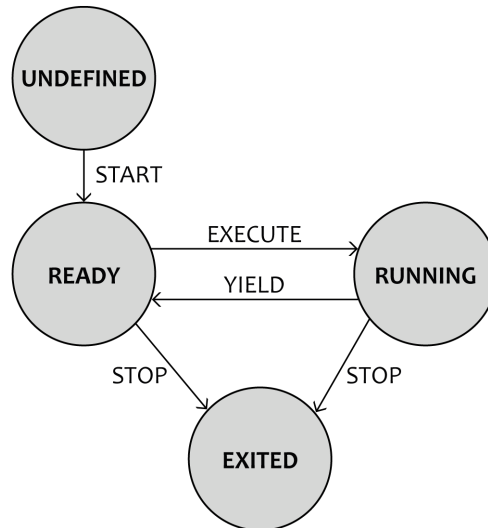


Figure 4.5: State chart of threads. Adapted from [59].

4.6 MEMORY ALLOCATION

Contiki provides three ways to allocate and deallocate memory [48]:

1. The memb memory block allocator - the allocator most frequently used;
2. The standard C library malloc heap memory allocator - the use of this allocator is discouraged. The reasons for it are going to be discussed further on this section;
3. The mmem managed memory allocator - almost not used.

In the rest of this section these three memory allocators are going to be present in more detail.

The memb Memory Block Allocator: In this allocator the memory blocks are allocated as an array of objects of constant size and are placed in static memory [60]. The memb offers a set of memory block management functions that are presented and explained in Table 4.3.

Function	Description
<i>MEMB(name, structure, num)</i>	Declares a memory block
<i>void memb_init(struct memb *m)</i>	Initializes a memory block
<i>void *memb_alloc(struct memb *m)</i>	Allocates a memory block
<i>int memb_free(struct memb *m, void *ptr)</i>	Frees a memory block
<i>int memb_inmemb(struct memb *m, void *ptr)</i>	Checks if an address is in a memory block

Table 4.3: The memb API functions.

Using the *MEMB()* macro a memory block is declared. This block is typically placed at the top of the C source file that uses the memory blocks. The arguments that should be passed into this function are [60]:

- *name* - identifies the memory block;
- *structure* - specifies the C type of the memory block;
- *num* - represent the amount of objects that the block accomodates.

Once the memory block is declared, it must be initialized through the *memb_init()* function. The argument that should be passed into this function, the pointer to a *struct memb*, identifies the memory location of the block [60]. After initialization, we can now use the *memb_alloc()* function to start allocating objects. This function returns a pointer to the allocated object if the operation was successful. If there are no available space to allocate the object, this function will return a NULL pointer. It is also possible to deallocate objects previously allocated by using the *memb_free()* function [60].

The malloc Heap Memory Allocator: The malloc allocator offers a set of functions to allocate and deallocate memory in the heap memory space [60]. These functions are presented and explained in Table 4.4.

Function	Description
<i>void *malloc(size_t size)</i>	Allocates uninitialized memory
<i>void *calloc(size_t number, size_t size)</i>	Allocates zero-initialized memory
<i>void *realloc(void *ptr, size_t size)</i>	Changes the size of an allocated object
<i>void free(void *ptr)</i>	Frees the memory

Table 4.4: The malloc API functions.

It is important to mention that most of the target Contiki platforms specify a small area of memory for the heap. The static memory allocations have a better performance in this type of constrained platforms, since dynamic allocations often incur in memory fragmentation. Allocation and deallocation of objects with different sizes may be problematic in these malloc implementations [60].

The mmem Managed Memory Allocator: This allocator provides a dynamic memory management quite similar to malloc. But it also offers a level of indirection in order to enable automatic defragmentation of the managed memory area [60]. In Table 4.5 the set of functions provided by this allocator are presented.

Function	Description
<i>MMEM_PTR(m)</i>	Provides a pointer to managed memory
<i>void mmem_init(void)</i>	Initializes the managed memory library
<i>int mmem_alloc(struct mmem *m, unsigned int size)</i>	Allocates managed memory
<i>void mmem_free(struct mmem *)</i>	Frees managed memory

Table 4.5: The memm API functions.

Every managed memory block is represented by an object of type *struct mmem*, as shown in Figure 4.6. The mmem library organizes the *struct mmem* objects in a list named mmemlist. In the *struct mmem* object, the *ptr* variable refers to the size of the allocated chunk in the contiguous memory pool reserved for the mmem library and the *size* variable indicates the amount of bytes that the memory block can store [60].

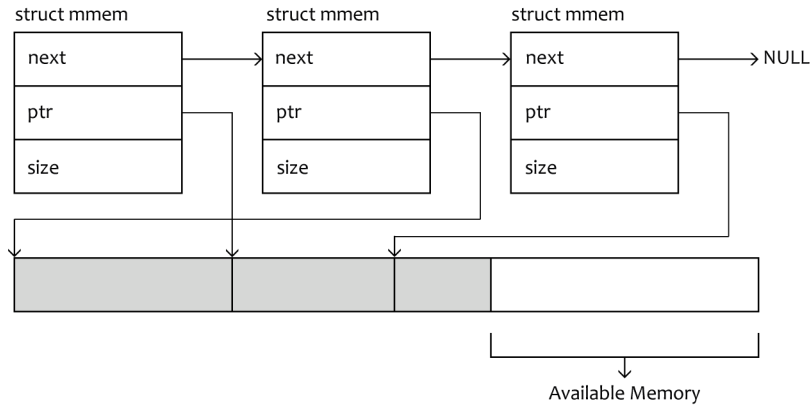


Figure 4.6: The managed memory allocator. Adapted from [60].

4.7 FILE SYSTEMS

The Contiki file system (CFS) defines an abstract API for reading/writing files and also for extracting directory contents. Contiki offers a set of file systems implementations that can be used with different kinds of storage in resource constrained devices. These file systems implement a subset of the CFS API, and two of them provide full functionality, namely [61]:

- CFS-POSIX - used in platforms running in native mode. This file system directly calls the POSIX file API that is provided by the host OS;
- Coffee - tailored to run in embedded platforms with flash memories or EEPROM.

Coffee was specially designed to make the file structure simple. For this purpose it makes use of extensions to the CFS API, in order to reduce the RAM memory usage. Coffee extends the CFS API with three functions [61]:

- *cfs_coffee_format()* - creates the initial empty file system;
- *cfs_coffee_reserve()* - reserves space for a file;

- *cfs_coffee_configure_log()* - configures a microlog file.

The microlog file structure is used to accomodate file modifications. This invisible file is a log structure and contains all the most recently written data. Whe this micro log eventually fills up, Coffee will merge the content of the original file and the micro log into a new file [61]. The main advantages associated to this file system are: requires very little metadata in RAM, and allows optimization on a per-file basis.

4.8 THE DYNAMIC LOADER

In order to support reprogramming and code swapping, Contiki provides dynamic linking and loading of modules. It is possible to load/unload a module, using one of two programming interfaces [62]:

- The executable Linkable Format (ELF) Loader - the dynamic loader will link and relocate ELF objects files into the Contiki system image. An ELF file consists of: a section for binary code (*.text*), a section for statically allocated data with pre-assigned values (*.data*), and a section for zero-initialized data (*.bss*);
- The native executable format of the host system when running a native platform - programs compiled in this format can be loaded by using the *dlloader_load()* function. This function is part of a module that can load software within systems, the *dlloader*. This module uses the dynamic linker provided by the host operating system.

4.9 LIBRARIES

4.9.1 TIMERS

The Contiki implements a set of timer libraries, which are responsible for: checking if a time period has passed, waking up the system from sleep mode at scheduled times, and scheduling real-time tasks [63]. Contiki has one clock module and a set of timer modules [48]:

- *timer* and *stimer* - are the simplest form of timers and are used to check if a time period has passed. They are used in low level drivers, since they can be safely called from interrupts. Regarding time resolution, the *timer* module uses system clock ticks and the *stimer* module uses seconds to allow longer periods;
- *etimer* - this module provides event timers. It is used in Contiki processes to wait for a time period while the rest of the system runs;
- *ctimer* - this module provides callback timers. It is used to schedule calls to callback functions after a time period. The *ctimer* is very useful in code that does not have a explicit process, like some protocol implementations. Therefore is used through the Rime stack implementation, to handle communication timeouts;
- *rtimer* - provides scheduling of real-time tasks. This timer module can preempt any running process, so that the real-time tasks can be executed within the scheduled time.

The Contiki clock module handles the system time, but also allows to block the CPU for short time periods. The timer libraries presented before are implemented with the functionality of the clock module as base [63].

4.9.2 LEDS API

LEDs are a simple but important tool to communicate with users or to debug purposes. The LEDs API functions are shown and explained in Table 4.6 [64].

Function	Description
<i>void leds_init(void)</i>	Initializes the LEDs driver
<i>unsigned char leds_get(void)</i>	Gets the status of a LED
<i>void leds_on(unsigned char ledv)</i>	Turns on a set of LEDs
<i>void leds_off(unsigned char ledv)</i>	Turns off a set of LEDs
<i>void leds_toggle(unsigned char ledv)</i>	Toggles a set of LEDs
<i>void leds_invert(unsigned char ledv)</i>	Toggles a set of LEDs

Table 4.6: The LEDs API functions.

4.9.3 THE SERIAL I/O API

In Contiki, the serial output is supported with the standard C library API for printing. On the other hand, the serial input relies on a Contiki specific mechanism [64].

Printing: The printing function - *printf()* - is supported with the linking of the standard C library provided by the compiler. After formatting the output string, *printf()* will simply call the *putchar()* function. The *putchar()* has an implementation part that depends on the specific hardware. This part must direct one byte at a time to the serial port [64].

Receiving: Contiki has an interface for serial line communication, available in *core/dev/serial-line.h* [48]. This interface has several functions [64]:

- *serial_line_init()* - initializes the process and ring buffer used;
- *serial_line_input_byte()* - when a character is ready to be read from the serial port, an interrupt is generated. The interrupt handler will then call this function, that will buffer data, until a line break is received;
- *serial_line_process()* - when a line break is received the *serial_line_input_byte* will poll the *serial_line_process()*;
- *serial_line_event_message()* - The *serial_line_process()* is responsible for broadcasting a *serial_line_event_message()* to all processes in the system.

4.10 COMMUNICATION

Since communication is a crucial concept in WSNs, Contiki implements communication as a service in order to enable run-time replacement. Therefore, it is possible in this OS to load multiple communication stacks at the same time [45]. The communication stack itself may be split into different services, as can be seen in Figure 4.7. This makes possible to replace individual parts of the communication stack at run-time.

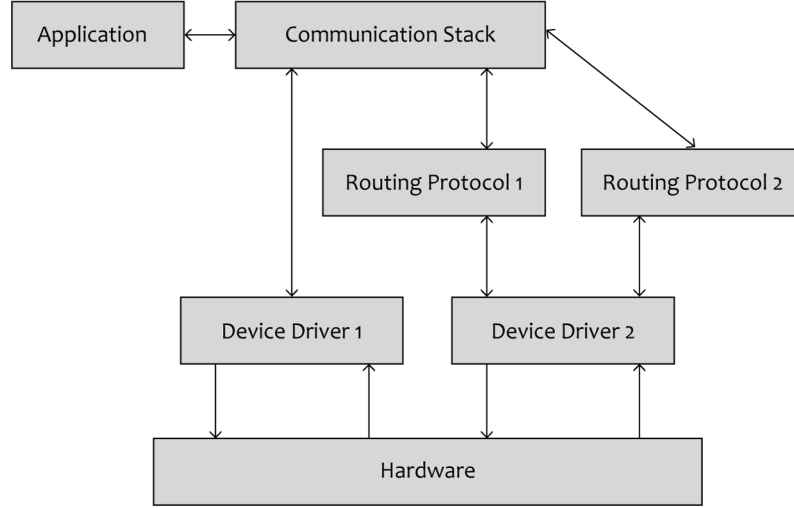


Figure 4.7: Contiki’s communication model. Adapted from [45].

The Contiki’s communication services use services mechanisms to communicate with each other and synchronous events to communicate with application services. They also use a single buffer for all communication processing. Hence, not needing data copying.

Basically when the radio receives a new data packet, the respective device driver will read this packet into the communication buffer. After that, it will call the upper layer associated service using the service mechanisms. After processing the packet headers the communication stack will post a synchronous event to the target application program. The application program will read the contents of the packet and, if necessary, will put a reply in the buffer before the communication stack takes control again. The communication stack, in turn, will add the respective header to the outgoing packet and will return the control to the device driver for the packet to be transmitted [45].

Contiki contains two communication stacks: uIP and Rime [48]. uIP is a small RFC-compliant TCP/IP stack that makes it possible for Contiki to communicate over the Internet. Rime is a lightweight communication stack designed for low-power radios. These two communication stacks are introduced with more detail in the next subsections.

4.10.1 UIP COMMUNICATION STACK

The uIP is a TCP/IP stack implemented in Contiki to support communication over the Internet. This stack was specially tailored to be used in resource constrained devices. Before uIP, the IP protocol

was seen as too heavyweight to be used in this constrained devices. The existing implementations of the IP protocol for general computers would need hundreds of kilobytes of memory, however a typical constrained device offers only a few kilobytes of memory [7]. Several non-IP stacks that could fit into this constrained memory were developed.

However, it was only in the early 2000's that this view was changed by the emergence of lightweight implementations of the IP protocol for smart objects, such as the uIP stack. With uIP stack it was possible to fit the IP protocol in constrained devices, without the need to remove essential IP mechanisms. Since its initial release, the uIP stack has become widely used in WSN applications [7] [54]. The uIP stack has the main following features [48]:

- It can only handle a single network interface;
- It implements the IP, ICMP, UDP and TCP protocols;
- It uses an event-based API in order to reduce code size and memory requirements;
- It uses a single global buffer for holding the packets and it has a fixed table for holding the connection state. This global buffer is large enough to contain one packet of maximum size;
- It is RFC compliant, but also IPv6 Ready Phase 1 certified;
- It is written in C language and it is fully integrated with the Contiki OS.

As stated, whenever a packet arrives, the device driver will place it in the buffer and the uIP stack will be responsible to call the target application service. Since the data in the buffer will be overwritten if a new data packet arrives, the application has to act immediately on the data by processing the packet or copying the data to its own buffer for later processing [48]. If a packet arrives when the application is processing the last incoming packet, the new packet must be queued by the network device or by the device driver. This means that uIP relies on the hardware when it comes to buffering.

Measurements show that the uIP stack provides very low throughput, particularly when communicating with a PC host [65]. However, the target constrained devices usually don't produce enough data to make the performance degradation a relevant problem [65].

4.10.2 RIME COMMUNICATION STACK

Besides the uIP communication stack, Contiki also supports the Rime communication stack that provides a set of lightweight communication primitives [48] [66]:

- Single-hop primitives;
- Multi-hop primitives - these primitives do not specify how packets are routed in the network. Instead, the upper layer protocol is invoked at every node to determine the next-hop neighbor of the packet. This offers the possibility to run arbitrary routing protocols using these multi-hop primitives. So, if a protocol or application running on top of the Rime stack needs a communication primitive that is not currently implemented, the application or protocol can implement it directly on top of the stack.

4.11 GLOBAL OVERVIEW

In this chapter all essential aspects of Contiki’s operation were presented and discussed. For a proper understanding, the Figure 4.8 shows a general overview of Contiki’s operation, regarding crucial mechanisms like event scheduling and process management.

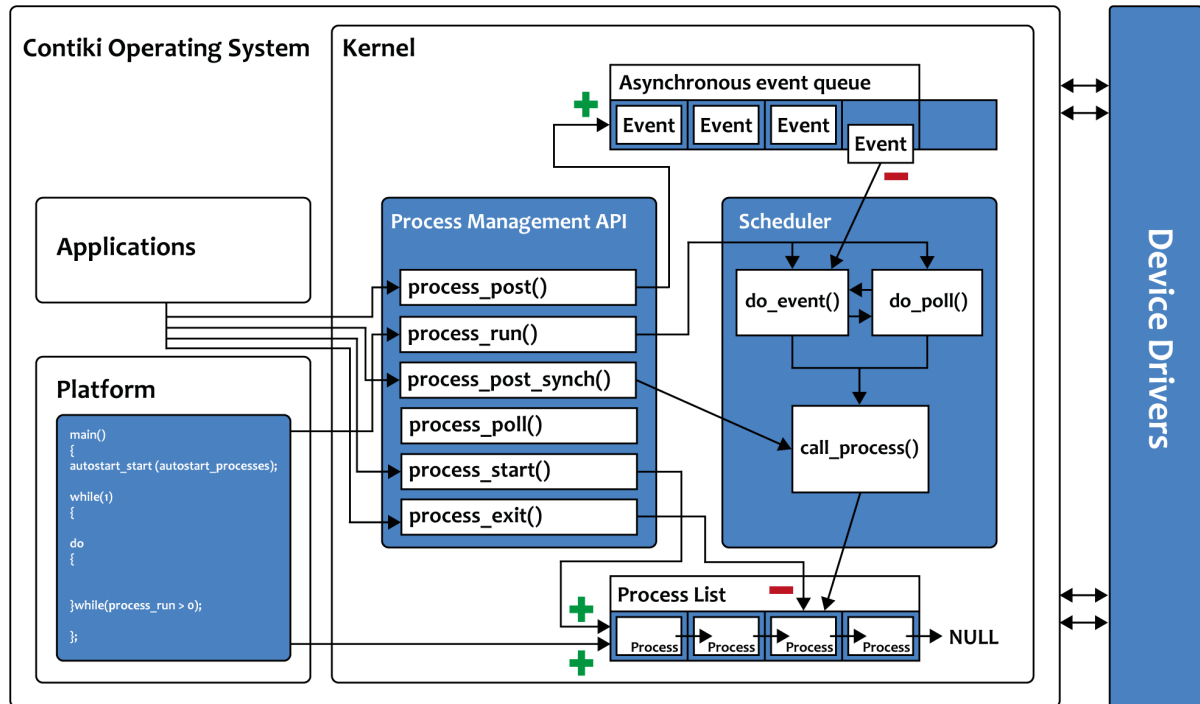


Figure 4.8: Contiki’s operation overview. Adapted from [57].

In this Figure it's represented one of the most important functions of Contiki: the *main()* function. This function is responsible for most of the system initializations. Among other responsibilities, it initializes the process management API, by calling the *autostart_start* function. All the processes listed in the *autostart_processes* will be initialized by calling that function. After the initialization phase the system enters in infinite loop. In this loop, the function *process_run()* is called periodically. If the function *process_run()* returns a value greater than "0", there are pending events in the asynchronous event queue or processes in need of poll. In that case, the scheduler will invoke the corresponding process, either in response to an event that has been posted to the process, or a poll that has been requested for the process. The Figure also presents the main functions of the process management API already introduced before.

IMPLEMENTATION

5.1 INTRODUCTION

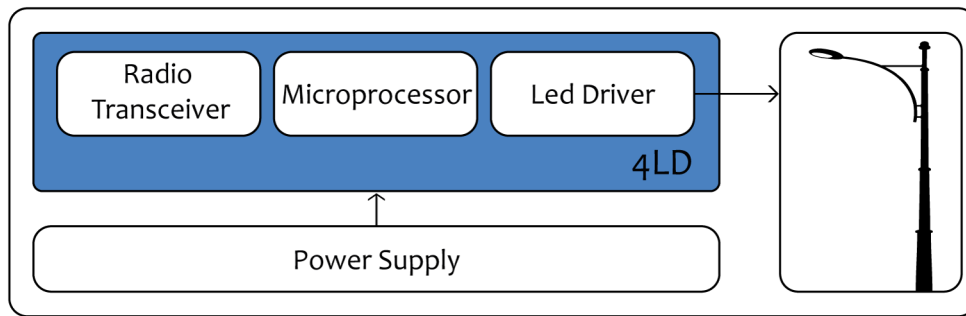


Figure 5.1: The 4LD platform as led controller.

Figure 5.1 presents one street lamp, powered by solar panels and controlled by the 4LD platform. The main components of the 4LD board are also depicted. In section 5.2 the 4LD platform is introduced with more detail.

The main goal of this implementation is establish a Wireless Mesh Network encompassing 4LD nodes that can be reached from the Internet. For this purpose we use the IPv6 support embedded in Contiki OS. After implementing the IPv6 Mesh Network, the goal is to send useful information over IP using application protocols like CoAP and OMA LWM2M.

As discussed, at least one gateway node that will translate and forward packets from one network to the other is required. In our experiments we used the Giore platform to implement the gateway node. In Section 5.2 the Giore platform is introduced with more detail.

5.2 HARDWARE

Two different hardware platforms were available for the experiments conducted for this thesis. One is the Giore platform, which was used to implement the gateway node. The other is the 4LD platform, which was used as part of the wireless sensor network.

5.2.1 THE GIORE PLATFORM

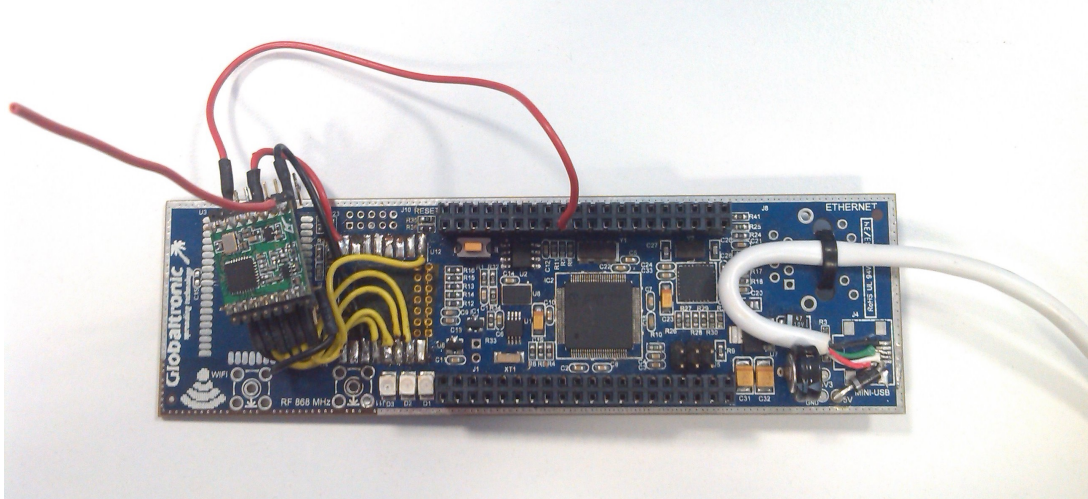


Figure 5.2: The Giore Board.

The hardware used to implement the gateway node is a module developed by Globaltronic S.A.. The module is equipped with a PIC32MX795F512L microprocessor, a TC1047AVNBTR temperature sensor and an external antenna. The radio interface for the 868 MHz ISM frequency is the RFM69HCW transceiver from Hope Electronics. This device offers the unique advantage of programmable narrow and wide-band communication modes and is also optimized for low power consumption while offering a high RF output power and channelized operation. This transceiver power consumptions are: 0.1 uA in Sleep Mode, 1.2 uA in Idle or Standby Modes, 16 mA in Rx Mode and 130 mA in Tx Mode, assuming a maximum output power of 20 dBm.

The PIC32MX microprocessor has 524KB of ROM, and includes 12KB of Boot Flash and 128KB of RAM. The PIC32 is a 32 bit microcontroller with a maximum operating frequency of 80 MHz. The MCU also includes a UART interface, which can be used to communicate over a serial link. Besides UART, it also has a SPI interface, which is strictly necessary to communicate with the RF transceiver. A picture of the platform can be seen in Figure 5.2.

A Microchip ICD3 development tool is used for programming this platform. In the experiments reported on this thesis the node was powered using the USB interface, but it can also be powered using an external power supply. The Giore platform also encompasses expansion slots for all the unused ports of the microcontroller. This allows attaching new modules and devices to the platform, offering new features. The platform features three LEDs, one green, one orange and one red, which mostly serve as visual flags for debug purposes.

The Giore platform is connected through SLIP (Serial Line over IP) to a computer running InstantContiki virtual machine. The computer has an Intel Core i3 processor and 4GB of RAM, with

only 1GB available for the virtual machine. This is more than enough for our purposes, and it should not force any restrictions on our solution. We will discuss the gateway implementation in more detail in Section 5.5.

5.2.2 THE 4LD PLATFORM

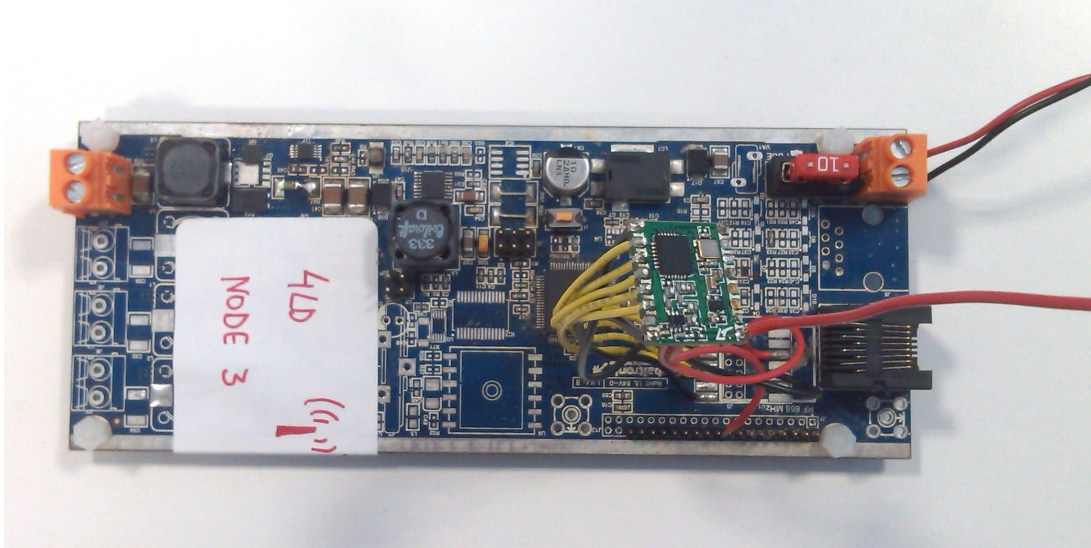


Figure 5.3: The 4LD Board.

The 4LD platform, as mentioned before, was also developed by Globaltronic S.A. and is used to control street and industrial lighting systems. A picture of the platform can be seen in Figure 5.3. The board is equipped with a PIC24FJ128GA308 microprocessor, a TC1047AVNBTR temperature sensor, a ST Microelectronics LIS2DHTR accelerometer and, also, the Hope Electronics RFM69HCW RF transceiver. One of the most important features of this board is the 4 channels available to drive LED lights in current mode.

The PIC24 is a 16 bit microcontroller with a maximum operating frequency of 32 MHz. It encompasses 128KB of ROM and only 8KB of RAM. The MCU includes the UART and SPI interfaces as well.

The Microchip ICD3 is also used as the core programming tool. The 4LD was powered using an external power supply. The platform also features one red LED, which mostly serves as a visual flag for debug purposes. This LED will be used to test the implementation using CoAP as application protocol.

5.3 PORTING THE HARDWARE TO CONTIKI OS

WSNs are highly application dependent and there are lots of available platforms. Each platform has its own set of sensors and may have different processors or radio transceivers. Since a WSN might

be composed of different hardware platforms which ideally have to run the same OS, it is clear to see why portability has become a crucial requirement for WSN operating systems.

Contiki's approach to the portability issue is limiting the abstraction provided by the system to just the basic: CPU multiplexing, event handling and support for loading programs and services. All the remaining abstractions are provided by the libraries that have nearly full access to the underlying hardware.

In our case we have two different platforms and the two have to run the Contiki OS. Thus, the first part of this thesis work was porting these two platforms to Contiki. Since Contiki doesn't offer support for the RFM69H transceiver, the second part was the development of the RF transceiver device driver. In the next subsections all the work needed to port the Giore and the 4LD platform, and also the development of the RF device driver, will be described.

In theory, porting Contiki to a platform requires only several modifications depending on the underlying hardware. However, in practice, it might be difficult to debug low-level hardware issues.

5.3.1 A GENERAL PORT

Before presenting the specific details of the Giore and 4LD ports, it is important to discuss first the essential steps for porting a new platform to Contiki. In Figure 5.4 we can see the Contiki's OS directory structure. There are basically two relevant directories when porting a new platform: the *platform* and *cpu* directories.

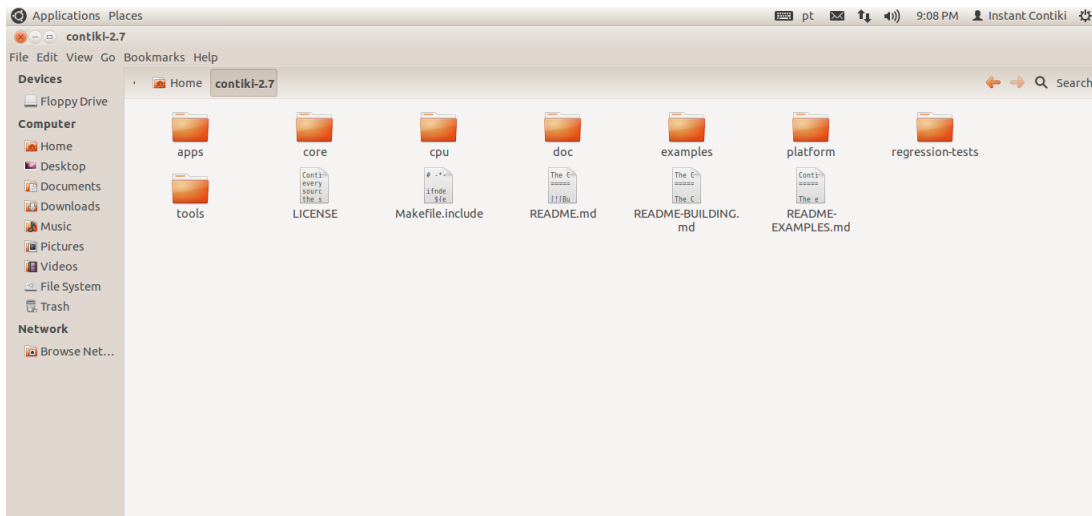


Figure 5.4: Contiki's directory structure.

The first thing to do is to create a subdirectory in the *platform* directory. In some ports it may be easier to copy and modify the files from */platform/native*, which contains the simplest Contiki port. In other ports it might be more useful to use ports from platforms that are similar to the platform to be ported, whenever available. For example in the case of the Giore board the */platform/seedeye* directory was used, since it contained the port to the SEED-EYE Board (which is similar to the Giore because it uses the same CPU - PIC32MX). The SEED-EYE port, which includes the PIC32 port to Contiki, was developed by the Networks of Embedded Systems Group of the Scuola Superiore Sant'Anna [67]. The CPU specific code is found in */cpu/cpuname*, which in this case is the */cpu/pic32* directory.

In order to integrate the new platform into Contiki some important files need to be created in the */platform/platformname* directory, such as:

- *contikiconf.h* - contains all platform specific configurations, such as: C types, compiler options, clock configurations, uIP configurations, pin configurations, etc;
- *Makefile.platformname* - contains platform specific options required to compile the platform. It usually defines the files containing low-level code, the type of processor used and the path to its source. Basically, this makefile usually specifies which files are to be compiled and includes *Makefile.include*, which includes all the other options needed;
- *contiki-platformname-main.c* - initializes the hardware and then schedules and runs the defined processes.

Besides the files in the */platform/platformname* directory, for every new type of CPU we must create a new subdirectory in the *cpu* directory. Inside this subdirectory some files need to be created, such as *clock.c*, which contains the clock driver functions: *clock-init()*, *clock-delay()* and *clock-time()*. It also contains a makefile to take care of the cross-compiler rules and specifies the path where to look for source files. In our case, the Giore platform uses the PIC32MX microprocessor and the Contiki OS already had this port implemented in the *cpu* directory. However, it was necessary to make some modifications to the initial port for proper integration. We will describe these modifications with more detail later in this section. In the case of the 4LD platform that uses the PIC24FJ microprocessor, it was necessary to write all the CPU port, because Contiki didn't offer support for the PIC24. The PIC24 port will also be described later in this section.

After creating all the previous files and directories, the next step to worry about is communication. We must ensure that the proper interface between uIP and low-level hardware code exists. In other words, we need to write the device driver for the platform's transceiver. The radio device driver specific code will be located in the directory */platform/platformname/dev/radioname*. In our specific case, it was necessary to write all the low level functions to interact with the RFM69H transceiver and to adapt the device driver to both platforms used in this thesis. Later in this section all work required to develop this device driver is described.

After all this is done, we should be able to successfully upload the Contiki OS into the new platform. In practice, beyond what was mentioned earlier, it may be necessary to add other features to the port, such as sensor support, for example. The code for the general sensor implementation is found in */core/lib/sensors.c*. The low-level code for controlling sensors should be located in a different directory: the */platform/platformname/dev* directory (usually).

5.3.2 PORTING THE GIORE PLATFORM

Porting to the Giore platform implied making some changes to the SEED-EYE port. However, no low-level code changes had to be done. Basically, the work done was towards system integration, in particular fixing makefiles and performing modifications to several files in order to provide higher level functionalities.

As stated, the Microchip PIC32 was, at the time this project started, already supported by Contiki OS. However, it was necessary to make some modifications in the SPI libraries because, due to some revisions, SPI register names were not up-to-date.

The resulting Giore port is complete and incorporates all the functionalities of a Contiki system. The code for the port can be found in */platform/giore* and in */cpu/pic32* directories. If we look into */platform/giore*, we will find the platform part of the port. Below is the explanation of the main files we will find therein.

Directory	File	Description
<i>/platform/giore</i>	<i>contiki-conf.h</i>	Hardware configuration file. The network stack settings are also defined here. The network stack configuration is an important part since we choose to have a WSN based on IPv6. Later, we will explain in more detail the selected drivers for the network stack.
<i>/platform/giore</i>	<i>contiki-giore-main.c</i>	System starter with <i>main()</i> function. It also starts most of the Contiki services.
<i>/platform/giore</i>	<i>init-net.c</i>	Network initialization for the Giore port.
<i>/platform/giore</i>	<i>Makefile.giore</i>	Settings needed to compile the Giore platform.
<i>/platform/giore/dev</i>	<i>leds-arch.c</i>	The architecture-dependent implementation of the Contiki's LED API. The microcontroller ports that drive the LEDs are defined and initialized with logic state 'o'.
<i>/platform/giore/dev</i>	<i>temperature-sensor.c</i>	The architecture-dependent implementation of the temperature sensor. It configures the AD converter to use 10 bits. It also has a function to obtain the temperature value in degrees Celsius, using the ADC and the equation provided by the TC1047AV datasheet.
<i>/platform/giore/dev/rfm69</i>	<i>rfm69h.c</i> <i>rfm69h.h</i> <i>rfm69h_arch.h</i>	RFM69H radio device driver. This will be explained in more detail later in this section.

Figure 5.5: Giore Port main files.

The *main()* function for the specific platform is defined in the *contiki-giore-main.c* file. It is in this function that the initialization of the drivers, processes and libraries necessary to the proper functioning of Contiki is realized.

The *main()* function calls the *pic32_init()* function, defined in the file *pic32.c* in the directory */cpu/pic32*, to set the mode operation of the quartz oscillator, which generates the clock signal used by the microcontroller timers with an oscillation frequency of 80 MHz. Then it also calls the *clock_init()* function, defined in the file *clock.c*, which will set Timer 1 to generate an interrupt at each 0.9766 ms. This period will be the lowest operating system unit of time in the platform. For each interrupt generated, Timer 1 will increase the variable count to keep the number of ticks that have elapsed since the operating system startup.

The function *main()* is also responsible to make the initialization of the LED API, so Contiki can be able to manage the platform LEDs. It also initializes the serial port and the process libraries. In order to unify the process of initialization of wireless communications a file called *init-net.c* was created on the Giore platform in the *platform* directory. This file defines the necessary steps to startup the radio transceiver driver, as well as defining the platform MAC and IPv6 addresses, after TCP/IP library startup.

After that, it also starts the process for the different OS timers (*etimer*, *ctimer* and *rtimer*), as well as the processes listed in the *autostart_processes*, calling the function *autostart_start*. After executing this initialization, the system progresses to an infinite loop and calls the function *process_run()* periodically. If the function *process_run()* returns zero, there are no pending events or processes in need of polling.

In order to generate a binary executable, the Microchip XC32 Compiler needs to be installed. Once this compiler is installed, we can use: `make TARGET=gione` to compile the application needed and it will generate one hex file with extension `.gione`. This hex file can be loaded into the platform using an ICD3 tool and the MPLAB IDE.

5.3.3 PORTING THE 4LD PLATFORM

Since at the time this project started there were no ports for the 4LD microprocessor in the community, the first phase of this port consisted in porting the Contiki OS to the Microchip PIC24 (PIC24FJ128GA308) MCU. The work developed is also relevant to other research groups, since Contiki will now support an additional processor.

This PIC24 port focused on the:

1. Analysis of the hardware capabilities offered by the Microchip PIC24;
2. Implementation of the timer routines required by Contiki using the existing hardware timers;
3. Implementation of a radio driver for the RFM69H transceiver. This part was important not only for this port, but was also necessary in the Gione Port. Therefore, this will be explained in more detail later in this section;
4. Development of some low-level drivers for peripherals like UART (e.g., for debugging purposes) and SPI (e.g., used by the radio transceiver communication with the MCU);
5. Development of other drivers such as the ones for LEDs and ADC.

The code for the 4LD port can be found in `/platform/leddriver` and in `/cpu/pic24`. If we look into `/cpu/pic24`, we will find the PIC24 port developed in this thesis. The description of the main files found there is presented in Figure 5.6.

The clock module, represented in Figure 5.6, has a crucial role to ensure the correct execution of the operating system, as it is responsible for the measurement of system time and defines a macro `CLOCK_SECOND`, which corresponds to one second in system ticks. A second on this platform corresponds to 64 system ticks, i.e., an elapsed second corresponds to 64 interrupts generated by microcontroller Timer 1. In the case of the 4LD platform the smallest unit of time is approximately 15.6 ms .

In the file `clock.c` we can find the following functions: `clock_init()` , `clock_delay()` , `clock_time()`, `clock_seconds()` and `clock_delay_usec()`. The `clock_init()` function is responsible for initializing the clock library, and must be called in the `main()` function. It will set the frequency of the microcontroller Timer 1, defining the minimum time resolution for the operating system.

The function `clock_time()` is responsible for reading and returning the global time value in system ticks when invoked. The `clock_seconds()` has the same features of `clock_time()`, only returning the system time in seconds. The operating system time values are stored in the global variable `count` for the system ticks and the variable `seconds` for the system seconds. The functions `clock_delay()` and `clock_delay_usec()` are responsible for providing the functionality to create delays in the system time.

At every 15.6 ms an interrupt is generated by the microcontroller and is handled by an interrupt service routine defined in the file `clock.c`. This ISR is responsible for increasing the overall system variable number of system ticks, check for timers `etimer` whose time has expired and each 64 ticks it will increase the global variable seconds.

Directory	File	Description
/cpu/pic24	clock.c	Contiki clock driver. It was configured to use PIC24 timer 1. Further on this section we will explain in more detail the main functions implemented on this file.
/cpu/pic24	pic24.c	PIC24 oscillator configuration and initialization functions.
/cpu/pic24	Makefile.pic24	Takes care of the cross compiler rules and definitions and also specifies the path where to look for source files.
/cpu/pic24/lib	pic24_clock.c	Clock interface functions for PIC24.
/cpu/pic24/lib	pic24_irq.h	Interrupt interface for PIC
/cpu/pic24/lib	pic24_spi.c	SPI low level functions for PIC24. Strictly necessary to communicate with the RF transceiver.
/cpu/pic24/lib	pic24_timer.c	Timer interface functions for PIC24.
/cpu/pic24/lib	pic24_uart.c	UART low level functions for PIC24. It includes UART configuration functions. Also implements test functions to read and write on the serial port. The UART was configured with one baudrate of 115200.

Figure 5.6: PIC24 port main files.

If we look into */platform/leddriver*, we will find the platform part of the port. Below is the explanation of the main files we will find therein.

Directory	File	Description
/platform/leddriver	contiki-conf.h	Hardware configuration file. The network stack settings are also defined here.
/platform/leddriver	contiki-leddriver-main.c	System starter with <code>main()</code> function. It also starts most of the Contiki services and libraries.
/platform/leddriver	init-net.c	Network initialization for the 4LD port. This function defines the necessary steps to the radio transceiver setup, as well as the definition of the MAC and IPv6 addresses.
/platform/leddriver	Makefile.leddriver	Settings needed to compile the 4LD platform.
/platform/leddriver/dev	leds-arch.c	The architecture-dependent implementation of the Contiki's LED API. The microcontroller ports physically attached to the platform LEDs as outputs are defined and initialized with logic state 'o'.
/platform/leddriver/dev	temperature-sensor.c	The low level functions to interact with the temperature sensor. It configures the AD converter and also has a function to obtain the temperature value in degrees Celsius.
/platform/leddriver/dev /rfm69	rfm69h.c rfm69h.h rfm69h_arch.h	RFM69H radio device driver. This will be explained in more detail later in this section.

Figure 5.7: 4LD Platform port main files.

In the file *contiki-leddriver-main.c* is implemented the *main()* function for the 4LD platform. In the this port, the function *main()* calls the *pic24_init()* function (defined in the file *pic24.c*) to set the mode operation of the quartz oscillator, which generates the clock signal used by the microcontroller timers (oscillation frequency of 32 MHz).

The function *main()* is also responsible to make the initialization of the LED API, so the Contiki can manage the platform LEDs. It also initializes the serial port, the process libraries and the process for the different OS Timers, as well as the processes listed in the *autostart_processes*. After executing this initialization, the system enters an infinite loop and calls the function *process_run()* periodically. If the function *process_run()* returns zero, there are no pending events or processes in need of poll.

In order to build an executable binary for the 4LD platform, the Microchip XC16 Compiler needs to be installed. Once this is concluded, we can use: *make TARGET=leddriver* to compile the application needed and it will generate one hex file with extension *.leddriver*. This hex file can then be loaded into the platform using an ICD3 tool and the MPLAB IDE.

5.3.4 RFM69H DEVICE DRIVER

After porting the two platforms into Contiki and having implemented the SPI low level functions, the next part of this work is to develop the RFM69H device driver for Contiki. First it's important to know the structure of a device driver for a radio in Contiki. After taking a look at [68], the radio device driver functions that we need to implement are:

```

/* Prepare the radio with a packet to be sent */
int(* prepare)(const void *payload, unsigned short payload_len)

/* Send the packet that has previously been prepared */
int(* transmit)(unsigned short transmit_len)

/* Prepare & transmit a packet */
int(* send)(const void *payload, unsigned short payload_len)

/* Read a received packet into a buffer */
int(* read)(void *buf, unsigned short buf_len)

/* Perform a Clear-Channel Assessment (CCA) to find out if there is a packet
   in the air or not */
int(* channel_clear)(void)

/* Check if the radio driver is currently receiving a packet */
int(* receiving_packet)(void)

/* Check if the radio driver has just received a packet */
int(* pending_packet)(void)

/* Turn the radio on */
int(* on)(void)

/* Turn the radio off */
int(* off)(void)

```

Listing 3: Radio Driver struct reference.

The file *rfm69.c* located at */platform/platformname/dev/rfm69* is the main file of the RFM69H radio driver developed and there we can find the implementation of the low level functions mentioned before. In Figure 5.8 a scheme explaining the developed device driver operation is presented. The *RFM69H_ISR()* is the interrupt service routine. This interrupt routine acts whenever the radio FIFO has some bytes that need to be read. When this happens the *rfm69h_process* is polled to read the packet received.

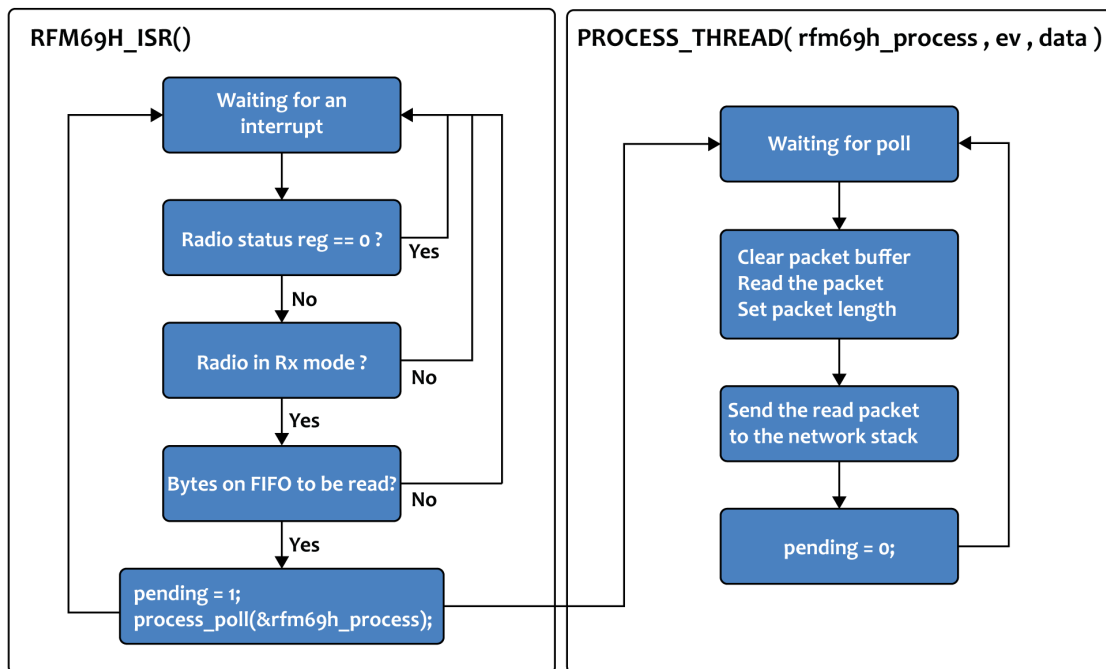


Figure 5.8: Radio device driver overview.

The RFM69H transceiver initialization is also an important part of this device driver. The RFM69H was configured with the following parameters:

1. Operation frequency of 868 MHz;
2. FSK Modulation;
3. Preamble of 5 bytes;
4. Sync word of 3 bytes.

An illustration of a RFM69 packet is shown below. The Preamble and Sync Word are added by the packet handler during the transmission process and removed during reception. The length byte is the first byte of the FIFO and indicates the length of the payload.

Preamble 5 bytes	Sync Word 3 bytes	Length byte	Payload
----------------------------	-----------------------------	-----------------------	----------------

Figure 5.9: RFM69H packet fields.

It is important to mention that, in the beginning, this device driver was implemented in such a way that the maximum packet that could be received/transmitted was 66 bytes, due to the RFM69H FIFO size. Later, it was necessary to receive/transmit packets with more than 66 bytes. The solution was to use on-the-fly FIFO readings and writes. Because of that it was necessary to rewrite the device driver functions.

5.4 THE NETWORK STACK

In Chapter 4 we described the main features and protocols available in the Contiki OS. Now these protocols are combined to implement our network stack. The Contiki stack is slightly different than the usual 5-layer model typically adopted in TCP/IP. Between the Physical and the Network layers, where usually is located the MAC, we have 3 different layers: Framer, Radio Duty-Cycle (RDC) and Media Access Control (MAC). The figure below shows the organization of these layers in the Contiki OS.

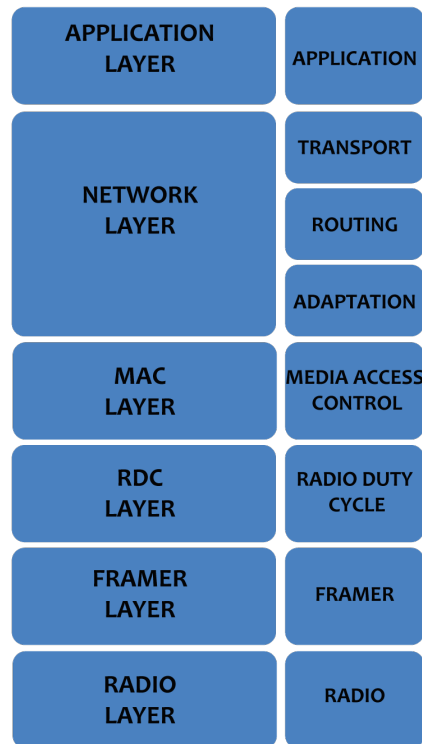


Figure 5.10: Contiki OS network stack.

Now we define which protocols will be used in each of the network stack layers. In Chapter 2 we already introduced in detail these protocols.

5.4.1 PHYSICAL, FRAMER, RDC AND MAC LAYERS

These layers can be defined through the global variables `NETSTACK_RADIO`, `NETSTACK_FRAMER`, `NETSTACK_RDC` and `NETSTACK_MAC` specified in the file *contiki-conf.h*.

Physical Layer: This layer corresponds to the RFM69H device driver presented in Section 5.2.

Framer Layer: This layer is responsible for creating frames with the data to be transmitted and for parsing the received data. The framer implementations in Contiki can be found in `/core/net/mac` directory. There are two types of framer layers: *framer-802154.c* and *framer-nullmac.c*. In our experiments we used the *framer-802154.c*. This framer type creates and parses frames compatible with the standard IEEE 802.15.4 [18]. The framer functions, located in the *framer-802154.c* file, are responsible for reading the data from the receive/transmit buffer, and insert/extract the same data into the *packetbuf* structure.

RDC Layer: This layer is responsible to determine the sleep period of the nodes. Basically, it decides when the packets must be transmitted and ensures that the node is awake when there are packets to be received from other neighbors. The RDC implementations are located in the `/core/net/mac` directory. There are several RDC protocols implemented in Contiki, such as: *contikimac.c*, *xmac.c*, *lpp.c*, *nullrdc-noframer.c*, *nullrdc.c* and *sicslowmac.c*. In our experiments we decide to use the *nullrdc* module to ensure that the transceiver remains always on. This module uses the functions from the framer layer for header creation and parsing. Despite what happens with the remaining RDC protocols, the *nullrdc* does not save energy and only acts as a pass-through layer. Other RDC implementations in Contiki, such as ContikiMAC, provides an efficient management of the energy spent. Due to the preliminary nature of this implementation we opted for not using such modules, thus simplifying the performance evaluation of our network.

MAC Layer: This layer takes care of addressing and retransmission of lost packets. The MAC implementations are located in the `/core/net/mac` directory. There are two types of MAC protocols implemented in Contiki: *csmac.c* and *nullmac.c*. In our case we opted to use the *nullmac* module. This MAC layer is a simple pass-through protocol that simply calls the appropriate RDC function.

5.4.2 NETWORK LAYER

At the network layer we adopted the 6LoWPAN protocol, to take advantage of the IPv6 protocol implemented by the Contiki's uIPv6 stack [48]. The network layer is configured with UDP and ICMPv6 protocols, and the routing protocol was the routing protocol for low-power and lossy IPv6 networks (RPL).

The 6LoWPAN implementation in Contiki corresponds to an adaptation layer called SICSLOWPAN. Whenever a Contiki device receives an IPv6 packet, the MAC layer will call the SICSLOWPAN to adapt the packets to be used by the IPv6 layer (being implemented by the uIPv6 stack) and when uIPv6 needs to send an IPv6 packet it also calls SICSLOWPAN to adapt it for the IEEE 802.15.4 standard frames.

Since the Contiki implementation of 6LoWPAN does not provide mesh-under or route-over mechanisms [69], the routing is provided by the RPL implementation called ContikiRPL. ContikiRPL forwards the packets to the uIPv6 stack, providing routing tables based on the different objective functions selected [70].

5.4.3 APPLICATION LAYER

In this work two different experiments using 4LD boards (as sensor nodes) and the Giore platform (as border router) are analysed and discussed. The first experiment employs the Erbium (Er) REST Engine and a CoAP implementation at the application layer. Erbium (Er) is a low-power REST engine developed for Contiki. It includes an embedded CoAP implementation and supports blockwise transfers and an observation mechanism.

In the second experiment we also use CoAP, but with the OMA LWM2M protocol, OMA LWM2M standard objects and IPSO objects in top of the CoAP implementation. LWM2M utilizes the CoAP as an underlying transfer protocol across UDP. The CoAP defines the message header, request/response codes, message options, and retransmission mechanisms. The LWM2M protocol only uses the set of features provided by CoAP. These two application protocols, the LWM2M and CoAP, were already introduced in Chapter 2.

To summarise, in the first experimental setup the network stack will follow the architecture shown in Figure 5.11.

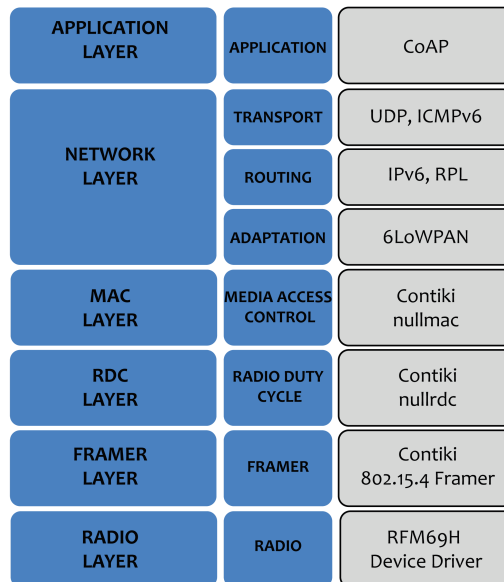


Figure 5.11: Network Stack used in the first experiment.

In the second experimental setup we added more layers on the top of the CoAP protocol, as can be seen in Figure 5.12.

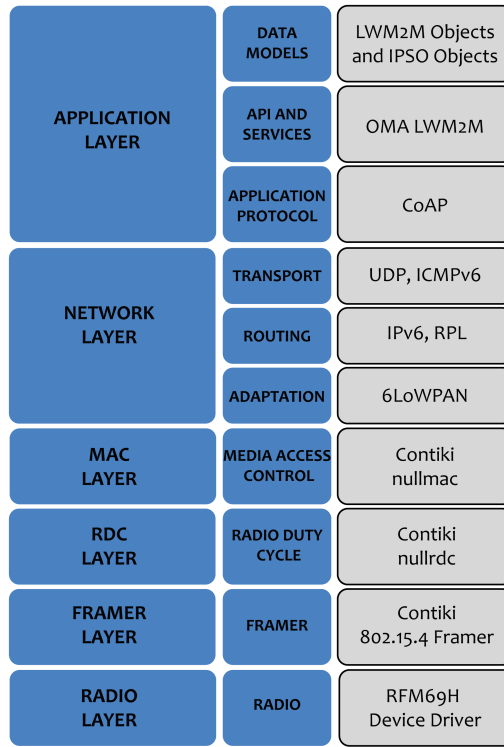


Figure 5.12: Network Stack used in the second experiment.

5.5 GATEWAY

In order to be able to access the WSN over the Internet using the IPv6 protocol it is necessary that one of the nodes in the network ensures the interconnection of the WSN network and the IPv6 network. As can be seen in Figure 5.13, the implemented gateway consists of two parts: the Border Router and the Base Station.

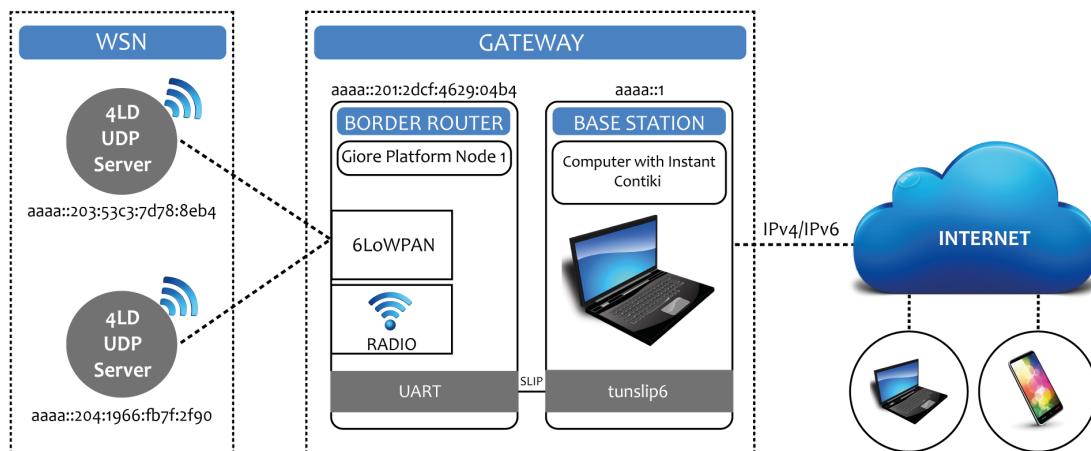


Figure 5.13: Overview of the gateway implemented in this thesis.

Border Router: The border router should be a device that has a high level of processing power and resources, in contrast to the WSN nodes. Therefore, we have chosen the Gioré platform to implement the border router, which is responsible for converting between IPv6 and 6LoWPAN. IPv6 is used in almost every communication link, but the final step from the border router to the sensor nodes use 6LoWPAN. The conversion is handled by Contiki. The border router is connected to the base station via a USB-dongle that provides a serial interface. Through a SLIP tunnel, the serial interface becomes the connection between the two parts of the gateway. The SLIP tunnel is a part of Contiki OS and it is used to encapsulate IP packets and send them over a serial link. The application connects to a serial port and creates a virtual network interface, which can be used by the base station as any other network interface.

Base Station: The base station is a personal computer that runs Instant Contiki, a Linux based operating system. Through the tool *tunslip6* available in the Contiki's directory *tools*, we can create a "tun" network interface on the base station with the address `aaaa::1` on the LAN.

Using these two parts together, the IPv6 packets can be routed between the WSN network and the Internet. Thus, the packets received over the SLIP tunnel will be forwarded and sent wirelessly to the 4LD nodes. The opposite direction is also possible: data received on the wireless interface will be forwarded and sent over the SLIP tunnel to the base station. Further ahead on this section we will explain with more detail the process required to create the "tun" interface.

5.5.1 THE GIORE AS BORDER ROUTER

The Gioré platform is programmed using the border-router solution available in the */examples/ipv6/rpl-border-router* directory. This solution is composed by 3 main files:

1. *border-router.c*
2. *slip-bridge.c* (It contains the callback function for processing a SLIP connection request)
3. *httpd-simple.c* (A simple Web server)

As previously aborded when presenting the RPL routing protocol, the 4LD nodes will form a DAG with the border router set as the root node. The border router will receive the prefix through the SLIP connection and will communicate it to the rest of the nodes in the RPL network.

Listing 4 presents a code snippet from the file *border-router.c*. In this piece of code the border router waits for the prefix to be set. Once it receives the prefix, the border router is set as the root of the DAG.

```

/* Request prefix until it has been received */
while(!prefix_set) {
    etimer_set(&et, CLOCK_SECOND);
    request_prefix();
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
}

```

```

dag = rpl_set_root(RPL_DEFAULT_INSTANCE,( uip_ip6addr_t *)dag_id);
if(dag != NULL) {
    rpl_set_prefix(dag, &prefix, 64);
    PRINTF("created a new RPL dag\n");
}

```

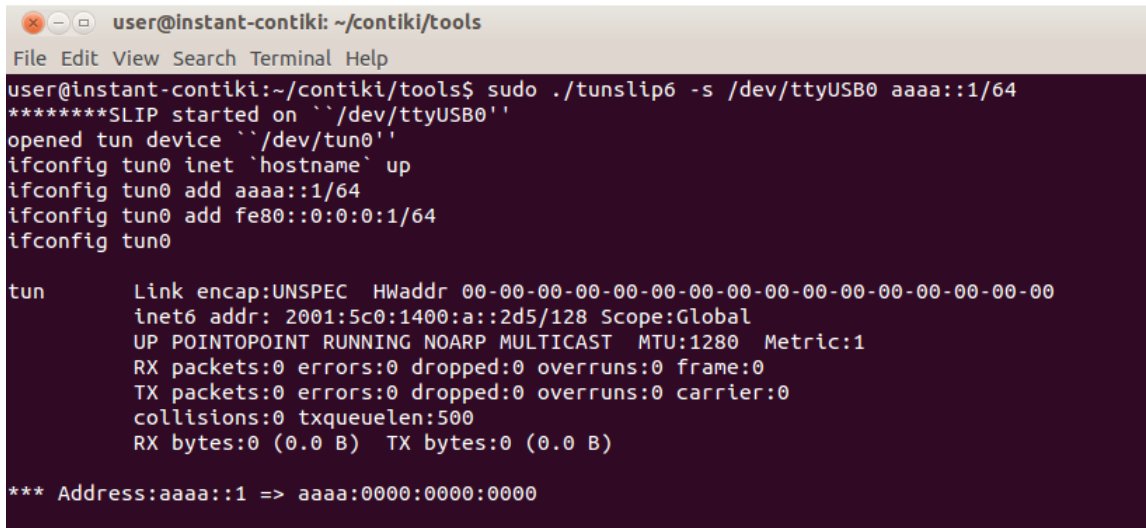
Listing 4: Code snippet from border-router.c.

By default, the border router hosts a simple Web page. However, this can be disabled through the macro `WEBSERVER`. The webpage is displayed when the IPv6 address of the border router is entered in the browser.

For test purposes, to check if our border router is working as desired, we programmed two 4LD nodes with the code *udp-server.c* available in the */examples/ipv6/rpl-udp* directory. At the end of this section we will show the results of pinging these nodes to test the connection between the WSN nodes and our base station.

5.5.2 THE SLIP TUNNEL

As mentioned earlier, in order to connect the WSN network and the base station, we need to create a SLIP tunnel, using the *tunslip* utility provided in Contiki. To create the SLIP tunnel, it is required to compile the *tunslip6* code, using the command: *make tunslip6*. Second, we make the connection between the WSN network and our base station, through the command: *sudo ./tunslip6 -s /dev/ttyUSB0 aaaa::1/64*. If this command is executed with success, the following output will be printed on the terminal:



```

user@instant-contiki: ~/contiki/tools
File Edit View Search Terminal Help
user@instant-contiki:~/contiki/tools$ sudo ./tunslip6 -s /dev/ttyUSB0 aaaa::1/64
*****SLIP started on `/dev/ttyUSB0'
opened tun device `/dev/tun0'
ifconfig tun0 inet `hostname` up
ifconfig tun0 add aaaa::1/64
ifconfig tun0 add fe80::0:0:0:1/64
ifconfig tun0

tun      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
        inet6 addr: 2001:5c0:1400:a::2d5/128 Scope:Global
        UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1280 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:500
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

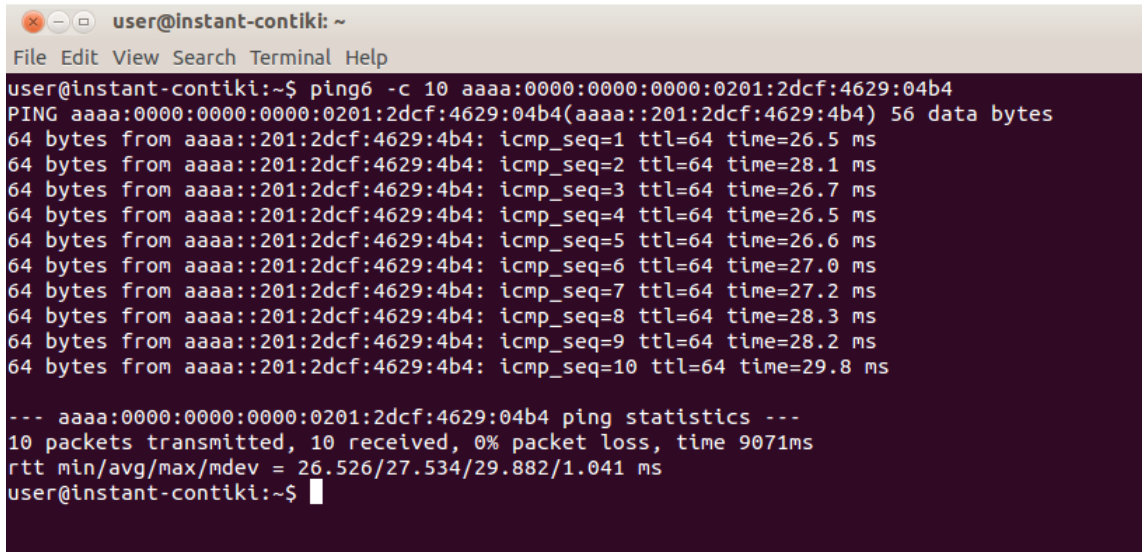
*** Address:aaaa::1 => aaaa:0000:0000:0000

```

Figure 5.14: Terminal print after creating the SLIP tunnel.

5.5.3 FUNCTIONAL TESTS

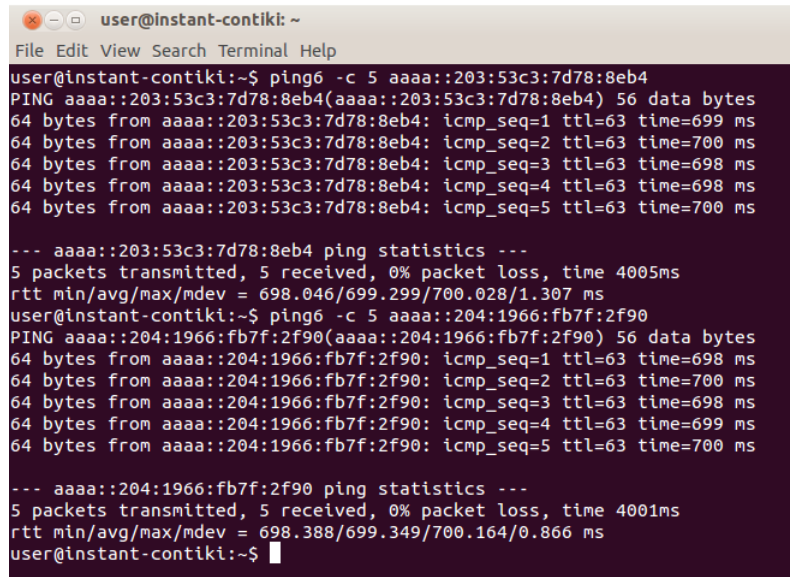
We can verify that the address of the border router has been set by using the ping command.



```
user@instant-contiki: ~  
File Edit View Search Terminal Help  
user@instant-contiki:~$ ping6 -c 10 aaaa:0000:0000:0000:0201:2dcf:4629:04b4  
PING aaaa:0000:0000:0000:0201:2dcf:4629:04b4(aaaa::201:2dcf:4629:4b4) 56 data bytes  
64 bytes from aaaa::201:2dcf:4629:4b4: icmp_seq=1 ttl=64 time=26.5 ms  
64 bytes from aaaa::201:2dcf:4629:4b4: icmp_seq=2 ttl=64 time=28.1 ms  
64 bytes from aaaa::201:2dcf:4629:4b4: icmp_seq=3 ttl=64 time=26.7 ms  
64 bytes from aaaa::201:2dcf:4629:4b4: icmp_seq=4 ttl=64 time=26.5 ms  
64 bytes from aaaa::201:2dcf:4629:4b4: icmp_seq=5 ttl=64 time=26.6 ms  
64 bytes from aaaa::201:2dcf:4629:4b4: icmp_seq=6 ttl=64 time=27.0 ms  
64 bytes from aaaa::201:2dcf:4629:4b4: icmp_seq=7 ttl=64 time=27.2 ms  
64 bytes from aaaa::201:2dcf:4629:4b4: icmp_seq=8 ttl=64 time=28.3 ms  
64 bytes from aaaa::201:2dcf:4629:4b4: icmp_seq=9 ttl=64 time=28.2 ms  
64 bytes from aaaa::201:2dcf:4629:4b4: icmp_seq=10 ttl=64 time=29.8 ms  
  
--- aaaa:0000:0000:0000:0201:2dcf:4629:04b4 ping statistics ---  
10 packets transmitted, 10 received, 0% packet loss, time 9071ms  
rtt min/avg/max/mdev = 26.526/27.534/29.882/1.041 ms  
user@instant-contiki:~$
```

Figure 5.15: Border router ping test.

We can also ping one of the other nodes in the network. In Figure 5.16 we are pinging the two 4LD nodes.



```
user@instant-contiki: ~  
File Edit View Search Terminal Help  
user@instant-contiki:~$ ping6 -c 5 aaaa::203:53c3:7d78:8eb4  
PING aaaa::203:53c3:7d78:8eb4(aaaa::203:53c3:7d78:8eb4) 56 data bytes  
64 bytes from aaaa::203:53c3:7d78:8eb4: icmp_seq=1 ttl=63 time=699 ms  
64 bytes from aaaa::203:53c3:7d78:8eb4: icmp_seq=2 ttl=63 time=700 ms  
64 bytes from aaaa::203:53c3:7d78:8eb4: icmp_seq=3 ttl=63 time=698 ms  
64 bytes from aaaa::203:53c3:7d78:8eb4: icmp_seq=4 ttl=63 time=698 ms  
64 bytes from aaaa::203:53c3:7d78:8eb4: icmp_seq=5 ttl=63 time=700 ms  
  
--- aaaa::203:53c3:7d78:8eb4 ping statistics ---  
5 packets transmitted, 5 received, 0% packet loss, time 4005ms  
rtt min/avg/max/mdev = 698.046/699.299/700.028/1.307 ms  
user@instant-contiki:~$ ping6 -c 5 aaaa::204:1966:fb7f:2f90  
PING aaaa::204:1966:fb7f:2f90(aaaa::204:1966:fb7f:2f90) 56 data bytes  
64 bytes from aaaa::204:1966:fb7f:2f90: icmp_seq=1 ttl=63 time=698 ms  
64 bytes from aaaa::204:1966:fb7f:2f90: icmp_seq=2 ttl=63 time=700 ms  
64 bytes from aaaa::204:1966:fb7f:2f90: icmp_seq=3 ttl=63 time=698 ms  
64 bytes from aaaa::204:1966:fb7f:2f90: icmp_seq=4 ttl=63 time=699 ms  
64 bytes from aaaa::204:1966:fb7f:2f90: icmp_seq=5 ttl=63 time=700 ms  
  
--- aaaa::204:1966:fb7f:2f90 ping statistics ---  
5 packets transmitted, 5 received, 0% packet loss, time 4001ms  
rtt min/avg/max/mdev = 698.388/699.349/700.164/0.866 ms  
user@instant-contiki:~$
```

Figure 5.16: 4LD Nodes ping test.

We can enter the address of the border router in the Web browser. The border router hosts a page that will be displayed on the browser as shown in Figure 5.17. In this webpage we have access to the routing table of the border router:

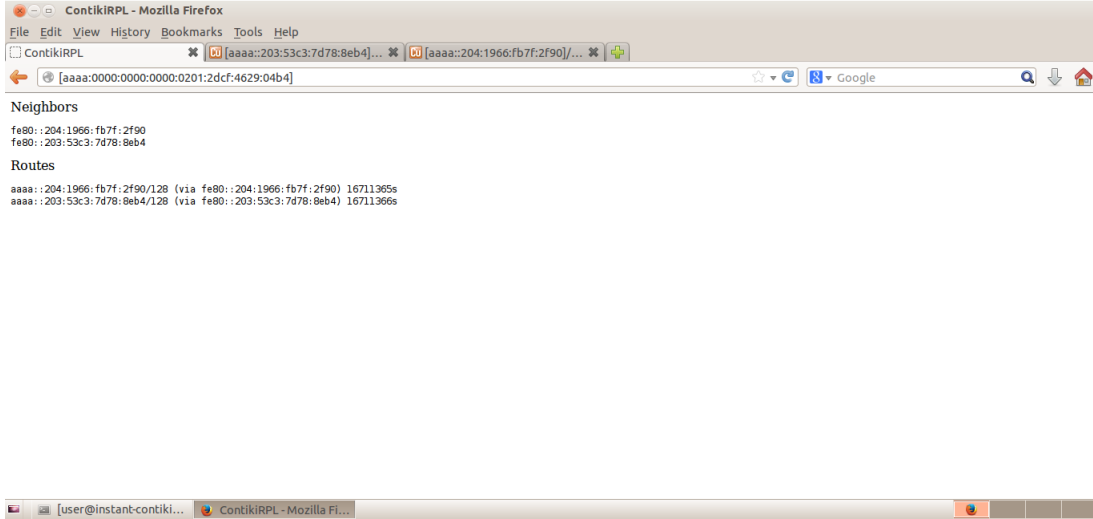


Figure 5.17: Neighbors and routes defined in the border router.

5.6 SENSOR NODE

As mentioned above, we implemented two experimental setups using sensor nodes based on 4LD boards and the Giore board as border router. For the first experiment we employed the Erbium (Er) REST Engine and CoAP. In this experiment we based our implementation in the example code *er-example-server.c* available in the *examples* directory of Contiki OS. This code is a RESTful server example showing how to use the REST layer to develop server-side applications. In the client-side application we used the Copper Plugin for Firefox. Copper is a Firefox extension allowing direct access to CoAP resources from a Web browser.

In the second experiment we also used CoAP, but with the OMA LWM2M protocol, OMA LWM2M objects and IPSO objects in top of the CoAP implementation. In this experiment we based our work in the Wakaama project source code. After making several modifications, we adapted the Wakaama code to run as an application in Contiki OS. Using this application, we developed an example of a client-side implementation that was used to program the 4LD nodes. In the server-side we used Leshan to interact with the LWM2M clients. Leshan is an LWM2M server implementation in Java. Basically, the LWM2M server manipulates resources on LWM2M clients using commands like Read, Write, Execute, Create or Delete. The LWM2M client may have any number of resources, each of which belongs to an object. In addition to the LWM2M standard objects we also implemented some IPSO objects, like the IPSO Temperature and IPSO Light Control.

These experiments are described in detail in the next subsections.

5.6.1 EXPERIMENTAL SETUP USING ERBIUM-COAP AND COPPER

An overview of the setup implemented using the Erbium REST Engine, CoAP implementation and the Copper Plugin can be seen in the following figure:

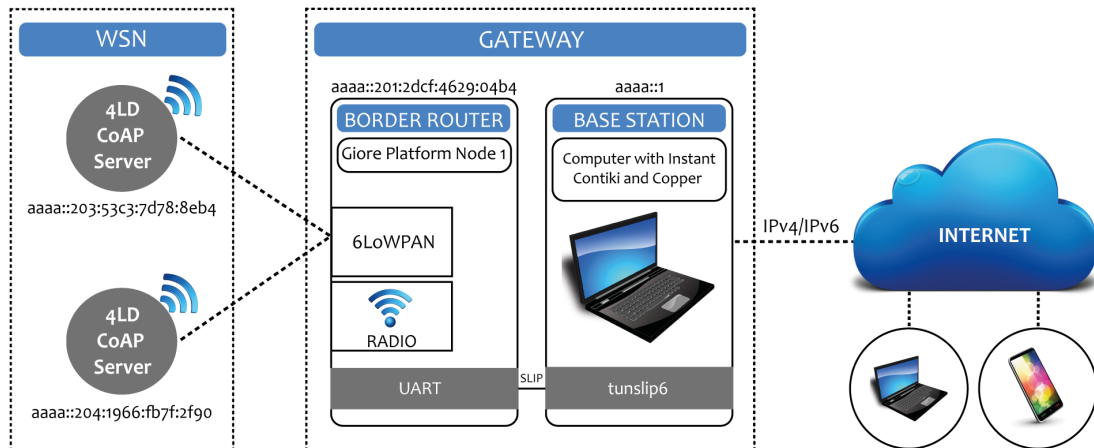


Figure 5.18: Overview of the experimental setup using Erbium-CoAP and Copper Plugin.

THE 4LD NODES AS COAP SERVERS

In this work, we have used Erbium-CoAP implementation that is provided by Contiki. The REST engine defined in this OS includes framework for developing both CoAP server and CoAP client applications.

CoAP resources are easily defined using predefined macros. For each resource, we simply have to define its name, path, interface description, resource type and the actual code to provide the data. For periodic resources, we also have to provide the period and for actuator resources, we have to provide the callback function performing the action. For example, the following code will instantiate a periodic resource, returning the temperature value with a period of 10 seconds and also an actuator resource that turns one light on/off:

```

/* Temperature sensor - periodic resource definition */
PERIODIC_RESOURCE(res_temperature,                                //Resource name
                  "title=\"Temperature_sensor\";obs",           //Resource attributes
                  res_get_handler,                               //Get handler
                  NULL,                                          //Put handler
                  NULL,                                          //Post handler
                  NULL,                                          //Delete handler
                  10 * CLOCK_SECOND,                             //Resource period
                  res_periodic_handler);                         //Periodic handler

/* Light on/off - actuator resource definition */
RESOURCE(res_light_control,                                       //Resource name
         "title=\"Light_Control\";rt=\"Control\"",              //Resource attributes
         res_get_handler,                                         //Get handler
         res_post_put_handler,                                    //Put handler
         res_post_put_handler,                                    //Post handler
         NULL);                                                    //Delete handler

```

Listing 5: Resource definitions.

All the resources are statically defined and the respective functions are registered when the REST engine (in the CoAP server) starts. Each resource has to implement a handler function with the name `[resource_name]_handler`. For example, when the CoAP client sends a GET request to the CoAP server for the URI `/temperature`, the `temperature_handler` function will be called by the REST engine. The handler implements actions for reading the temperature sensor and returning the temperature value. The CoAP response message is formatted according to the client requested format, which can be plain text, XML or JavaScript Object Notation (JSON).

A typical RESTful Web service application consists of one main C-file, the *er-example-server.c* available in the directory `/examples/setup-er-rest`. It contains one Contiki process that initializes the REST Engine and activates the resources, as can be seen in Listing 6. In the subdirectory *resources* the resource's files are available. The resource macros together with their handler functions are defined for each resource in these files.

```
#if PLATFORM_HAS_LEDS
    rest_activate_resource(&res_toggle, "actuators/led_toggle");
#endif

#if PLATFORM_HAS_LIGHT_CONTROL
    rest_activate_resource(&res_light_control, "actuators/light_on");
    rest_activate_resource(&res_light_dimmer, "actuators/light_dim");
#endif

#if PLATFORM_HAS_TEMPERATURE_SENSOR
    rest_activate_resource(&res_temperature, "sensors/temperature");
#endif

rest_activate_resource(&res_device_info, "device/info");
rest_activate_resource(&res_device_uptime, "device/uptime");
```

Listing 6: Resource activations.

In terms of resource discovery, the CoRE Link Format is generated automatically for all resources. Our handler for the `/.well-known/core` URI respects chunk-wise processing and generates the required substrings without exceed the size of the buffer provided by the REST Engine. The application developed in this thesis implements seven different resources, as can be seen in Figure 5.19.

RESOURCE URI	FUNCTIONALITY
<code>/.well-know/core</code>	Resource Discovery
<code>/device/info</code>	Information about the 4LD node
<code>/device/uptime</code>	Information about the seconds elapsed
<code>/actuators/led_toggle</code>	Actuator that turns the red led on/off
<code>/actuators/light_on</code>	Actuator that turns the 4LD light on/off
<code>/actuators/light_dim</code>	Actuator resource that dims 4LD light
<code>/sensors/temperature</code>	Resource that returns the temperature

Figure 5.19: CoAP resources implemented in the 4LD Server.

COPPER PLUGIN AS COAP CLIENT

The Copper (Cu) CoAP user-agent for Firefox browser installs a handler for the CoAP URI scheme and allows users to interact with Internet of Things devices. It includes several features, such as:

- URI handling for the CoAP scheme (address bar and links);
- Interactions using the GET, POST, PUT, and DELETE requests;
- Resource discovery;
- Blockwise transfers;
- Observing resources.

To use Copper we only need to enter in the browser address bar: `coap://[ipv6addressofthecoapserver]:udp_port`. After opening the location of the CoAP server a click on "Discover" button will retrieve the available resources from `/.well-known/core`. At a resource location (e.g., `coap://[aaaa::203:53c3:7d78:8eb4]:5683/actuator/light_dim`) we only need to use the buttons GET, POST, PUT, DELETE to perform the desired action. The response will then be displayed in the browser.

FUNCTIONAL TESTS

After entering the addresses `coap://[aaaa::203:53c3:7d78:8eb4]` and `coap://[aaaa::204:1966:fb7f:2f90]` in the browser address bar, and after clicking in the "Discover" button, we obtain the resources available in each server. 4LD nodes will respond with a CoAP message with the following payload:



Figure 5.20: Server response to the GET action on the resource `/.well-know/core`.

After receive this message, the following outputs will be displayed in Copper:

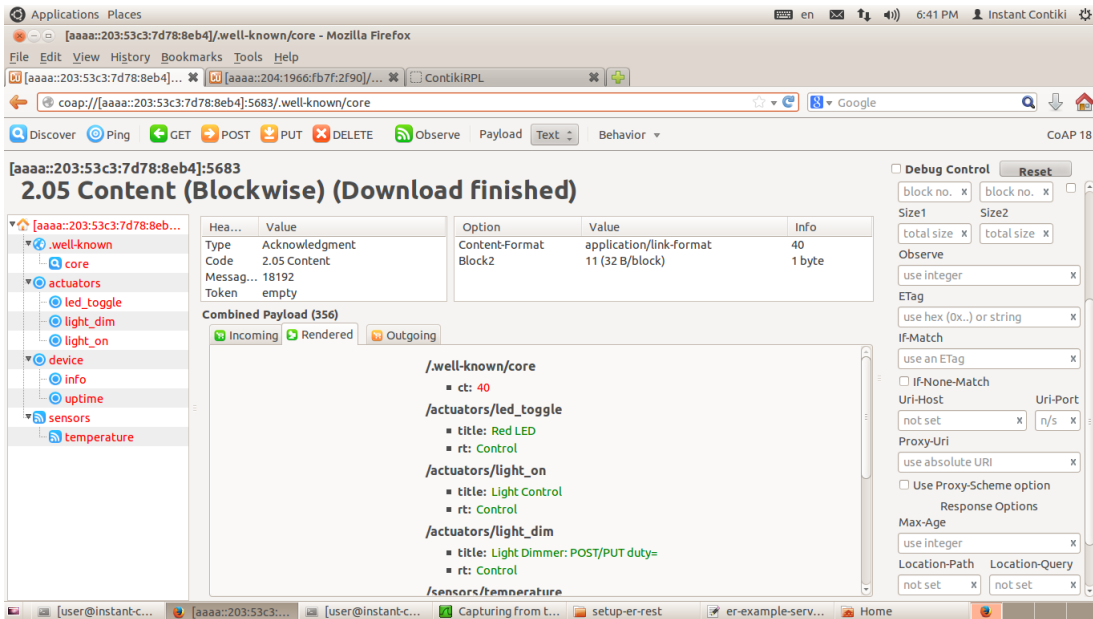


Figure 5.21: Copper output from 4LD Node 3.

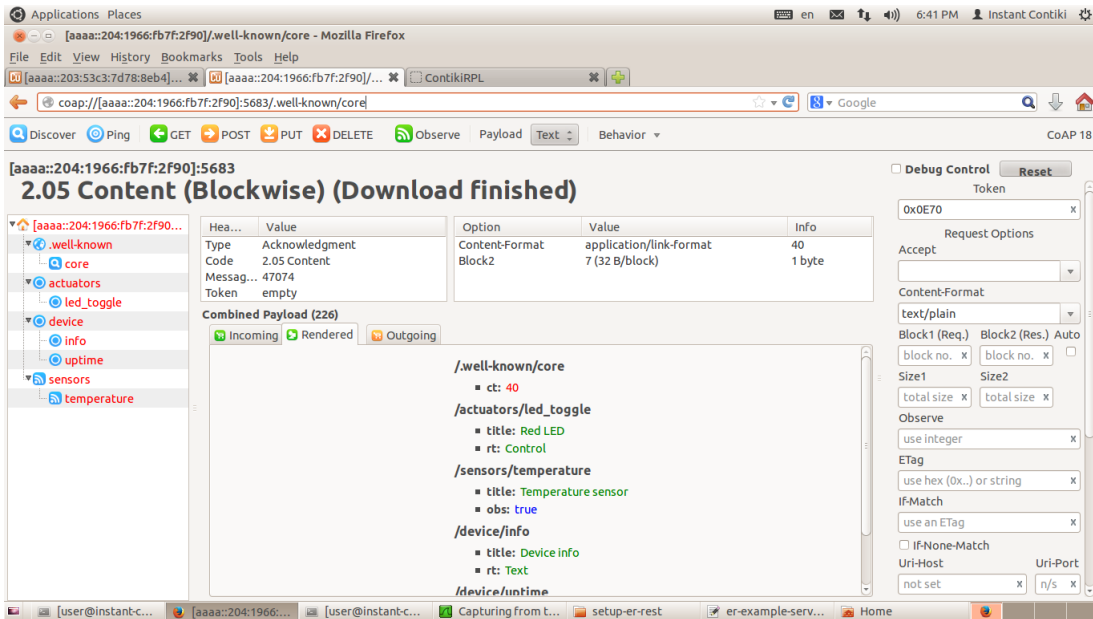


Figure 5.22: Copper output from 4LD Node 4. Since the 4LD LED channels are not mounted in this platform, the `light_on` and `light_dim` resources are not implemented.

After, we only need to use the buttons GET, POST and PUT to perform actions. An overview of the available actions in each resource are documented in Figure 5.23.

RESOURCE URI	METHOD GET
<code>/.well-know/core</code>	Retrieves all the resources implemented in the server.
<code>/device/info</code>	Retrieves the software and hardware versions.
<code>/device/uptime</code>	Retrieves the seconds elapsed since the platform has been attached.
<code>/actuators/led_toggle</code>	Retrieves the actual led state. It will retrieve '1' when the led is on, and '0' otherwise.
<code>/actuators/light_on</code>	Retrieves the actual light state. It will retrieve '1' when the light is on, and '0' otherwise.
<code>/actuators/light_dim</code>	Retrieves the actual PWM duty cycle. The initial value is 50%.
<code>/sensors/temperature</code>	Retrieves the actual temperature value. Is the only observable resource implemented.

RESOURCE URI	METHOD PUT
<code>/.well-know/core</code>	Method not allowed.
<code>/device/info</code>	Method not allowed.
<code>/device/uptime</code>	Method not allowed.
<code>/actuators/led_toggle</code>	Turns the led on and off.
<code>/actuators/light_on</code>	Turns the light on and off. We should sent '1' to turn on the light and '0' otherwise.
<code>/actuators/light_dim</code>	Changes the PWM duty cycle to the desired value. The values should be between 0-100.
<code>/sensors/temperature</code>	Method not allowed.

RESOURCE URI	METHOD POST
<code>/.well-know/core</code>	Method not allowed.
<code>/device/info</code>	Method not allowed.
<code>/device/uptime</code>	Method not allowed.
<code>/actuators/led_toggle</code>	Method not allowed.
<code>/actuators/light_on</code>	Turns the light on and off. We should sent '1' to turn on the light and '0' otherwise.
<code>/actuators/light_dim</code>	Changes the PWM duty cycle to the desired value. The values should be between 0-100.
<code>/sensors/temperature</code>	Method not allowed.

Figure 5.23: Available actions in each resource.

Hence, for example, to request the temperature we should go to the resource location `/sensor/temperature` and perform a GET action. Using Wireshark to capture the packets in the SLIP tunnel, Copper will send the following message to the CoAP Server requesting the temperature value:

+ Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0			
+ Raw packet data			
+ Internet Protocol Version 6, Src: aaaa::1 (aaaa::1), Dst: aaaa::203:53c3:7d78:8eb4 (aaaa::203:53c3:7d78:8eb4)			
+ User Datagram Protocol, Src Port: 43402 (43402), Dst Port: 5683 (5683)			
+ Constrained Application Protocol, Confirmable, GET, MID:8304			
01.. = Version: 1			
..00 = Type: Confirmable (0)			
.... 0000 = Token Length: 0			
Code: GET (1)			
Message ID: 8304			
+ Opt Name: #1: Uri-Path: sensors			
+ Opt Name: #2: Uri-Path: temperature			
+ Opt Name: #3: Block2: NUM:0, M:0, SZX:32			

0000	60 00 00 00 00 22 11 40	aa aa 00 00 00 00 00 00	'...."@
0010	00 00 00 00 00 00 00 01	aa aa 00 00 00 00 00 00 3..").
0020	02 03 53 c3 7d 78 8e b4	a9 8a 16 33 00 22 29 e8	..s..}x...3..").
0030	40 01 20 70 b7 73 65 6e	73 6f 72 73 0b 74 65 6d	@. p.sen sors.tem
0040	70 65 72 61 74 75 72 65	c1 01	perature ..

Figure 5.24: GET message to request the sensor temperature value.

The CoAP server will reply with the ACK message documented in Figure 5.25, in this case the temperature is 26 degrees Celsius:

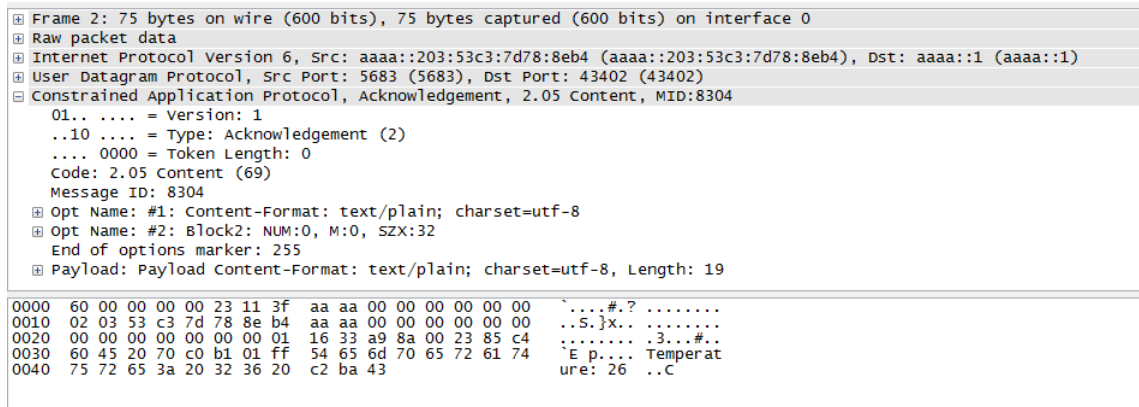


Figure 5.25: ACK message from the 4LD CoAP server with the temperature value.

In the next chapter, a more in-depth evaluation of this experiment will be conducted, regarding topics such as memory usage, round trip time and packet loss.

5.6.2 EXPERIMENTAL SETUP USING OMA LWM2M AND LESHAN SERVER

An overview of the setup implemented using the OMA LWM2M protocol, IPSO objects and the Leshan server can be seen in the following figure:

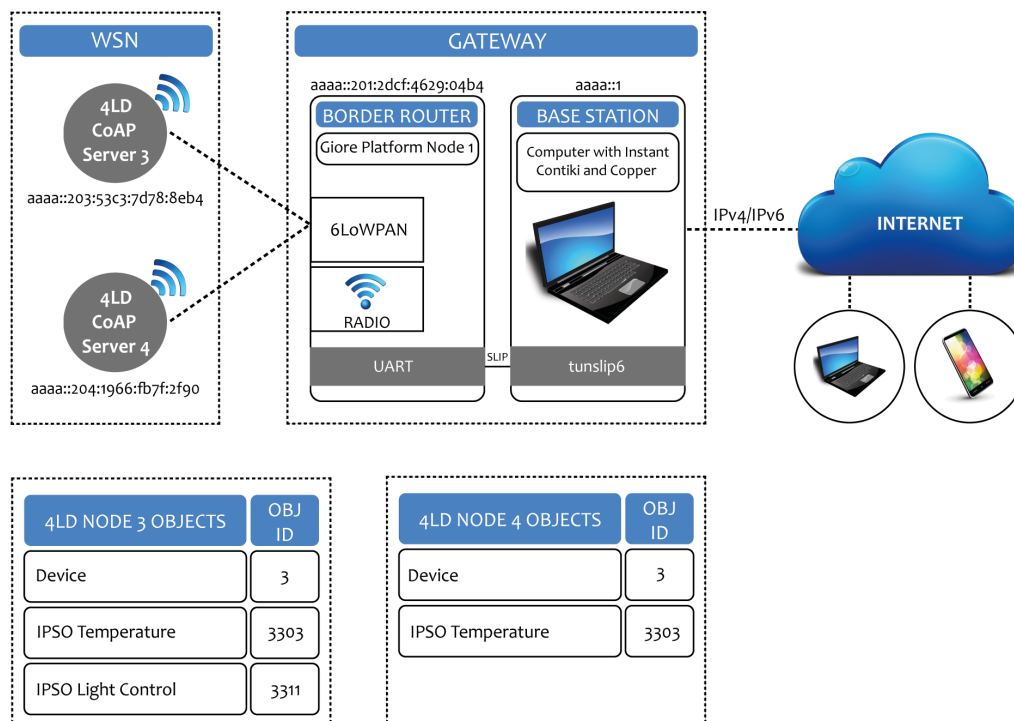


Figure 5.26: Experimental setup using OMA LWM2M overview.

THE 4LD NODES AS LWM2M CLIENTS

As mentioned, the LWM2M client implementation was based on the Wakaama project. The Wakaama project covers the LWM2M protocol, CoAP, and DTLS layers of the LWM2M protocol stack for three components: LWM2M client, LWM2M server and LWM2M bootstrap server. Wakaama is not a library but a set of files that must be built together with an application. It is written in the C language and designed to be portable on POSIX compliant systems. Two compilation switches are available: `LWM2M_CLIENT_MODE` and `LWM2M_SERVER_MODE`. Defining `LWM2M_CLIENT_MODE` enables the LWM2M client interfaces. Defining `LWM2M_SERVER_MODE` enables the LWM2M server interfaces.

In our case, we ported the Wakaama files to run as an application on top of Contiki OS and we implemented an OMA LWM2M client application to program the 4LD nodes. In the `/apps/lwm2m` directory the main files ported from Wakaama project are listed, as can be seen in the following figure:

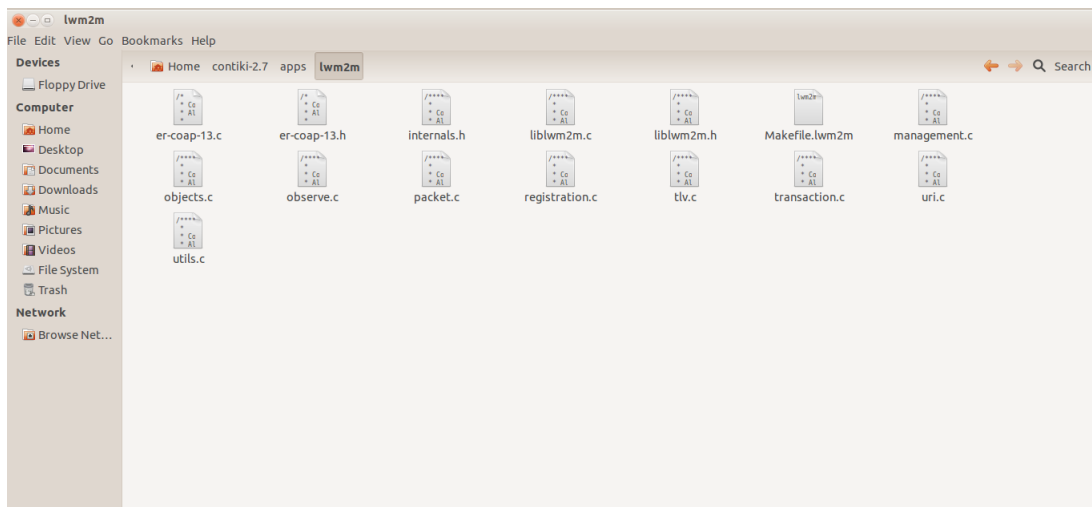


Figure 5.27: LWM2M implementation files.

It is worth mentioning that this LWM2M implementation uses dynamic memory allocation. Due to this feature, it was necessary to reserve memory space for the heap. Since 70% of the RAM was needed to store data memory from the program, we only have reserved 1024 bytes for the heap, because we need at least 2kB for the stack. This use of dynamic memory sometimes can cause memory fragmentation and can lead to unexpected crashes. So, this ported implementation is more a proof of concept than a perfect solution. Furthermore, in the Future Work Section we present some lines for further development which includes the rewriting of the LWM2M implementation to use static memory instead of dynamic, avoiding the memory fragmentation problem.

In the directory `/examples/LWM2M-Tests/client` all the files necessary to implement the OMA LWM2M Client are listed:

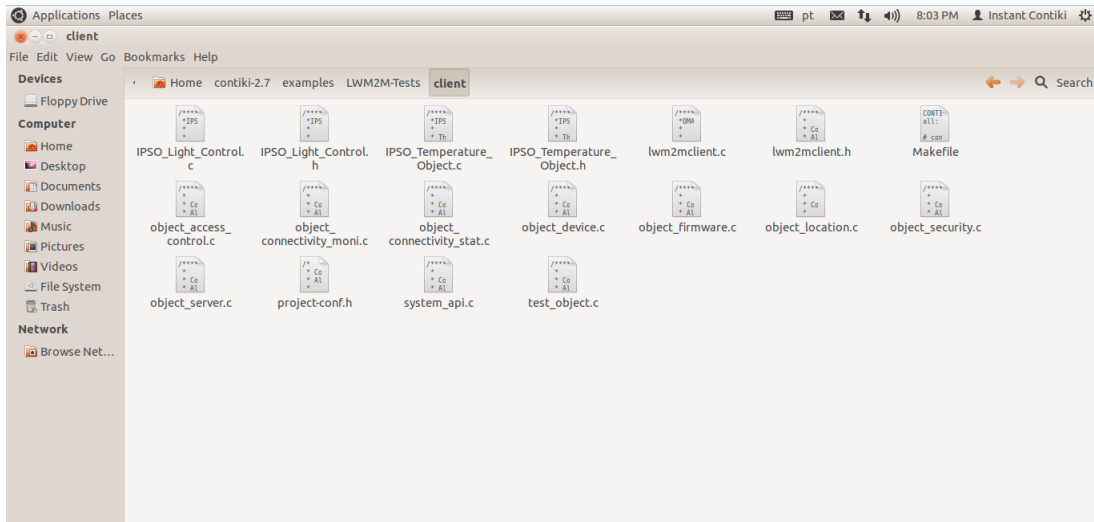


Figure 5.28: LWM2M Client side implementation files.

The main implementation file is the *lwm2mclient.c*. This file is responsible for creating the LWM2M objects and it also initializes the lwm2m library. Before it initializes the lwm2m library, it will wait 1 minute, so the platform can be able to configure all RPL routes. This *lwm2mclient* opens the UDP port 5683 and tries to register to a LWM2M server. In this case, it will make the registration in the Leshan server. The registration phase is explained in more detail furthermore on this Section.

The following files *object_access_control.c*, *object_connectivity_moni.c*, *object_connectivity_stat.c*, *object_device.c*, *object_firmware.c*, *object_location.c*, *object_security.c* and *object_server.c* are responsible to implement the OMA LWM2M standard objects for device management purposes, already presented in Chapter 2. The files *IPSO_Light_Control.c* and *IPSO_Light_Control.h* are responsible to implement the light control object from IPSO specification. This object is used to control a light source, as already mentioned before. The files *IPSO_Temperature_Object.c* and *IPSO_Temperature_Object.h* are also responsible to implement another IPSO object, but in this case the IPSO temperature object. These IPSO objects were already introduced in Chapter 2.

Due to memory constraints in the 4LD platform, it is not possible to implement all the objects mentioned above. The Makefile makes it possible to choose which objects are going to be available in LWM2M client. In both 4LD Nodes only the mandatory objects from OMA LWM2M standard are defined: LWM2M Security, LWM2M Server and Device.

LESHAN AS LWM2M SERVER

Leshan is an OMA LWM2M implementation in Java. The Leshan project provides a complete infrastructure for building IoT solutions, such as: a device management server library, a device management client library and a device management server with a Web user interface. In this specific case we will only use the device management server with a Web user interface to test our LWM2M clients.

To use this device management server we should run the following command: `java -jar ./leshan-standalone.jar`. Afterwards, to use the server user interface it is only necessary to enter in the browser address bar: `http://localhost:8080`. Leshan provides a very simple user interface to list the connected clients and interact with the clients resources.

The first thing that the LWM2M Client should do is to register in the Leshan Server. For that it will send the following message:

Figure 5.29: Message sent from the 4LD LWM2M Client (Node 4) to make the registration in the Leshan Server.

This message contains the name of the client, which in this case is "LED DRIVER - NODE 4". It also contains the Objects IDs defined in the client. The Leshan server will reply with the following message:

Figure 5.30: Acknowledge sent from Leshan to the 4LD Client (Node 4) with the Registration ID.

This is an acknowledgement message with the registration ID. The LWM2M Client should then send a registration message with the Registration ID each 300s, otherwise the Leshan Server will deregister the client. This message is shown in Figure 5.31.

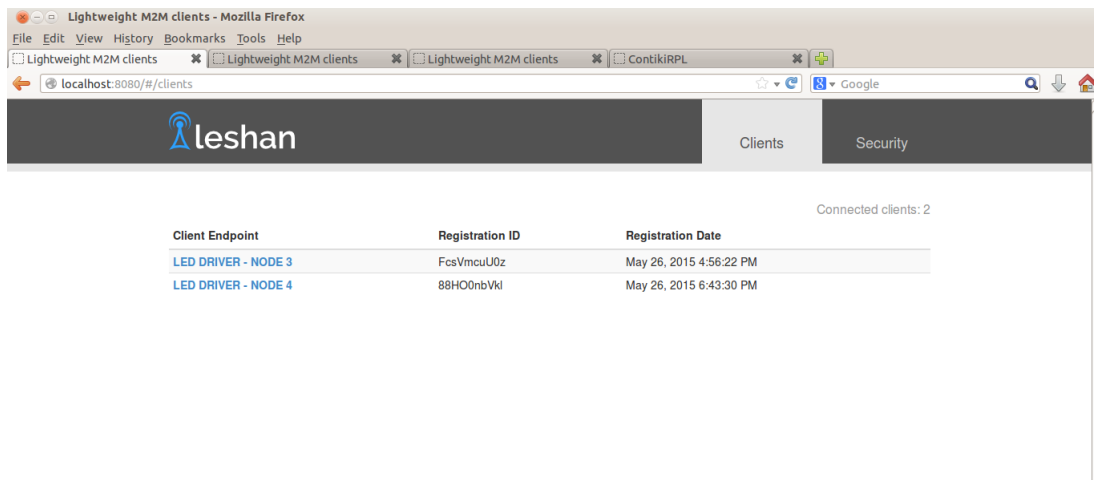
```

Frame 89: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
Raw packet data
Internet Protocol Version 6, Src: aaaa::203:53c3:7d78:8eb4 (aaaa::203:53c3:7d78:8eb4), Dst: aaaa::1 (aaaa::1)
User Datagram Protocol, Src Port: 5684 (5684), Dst Port: 5683 (5683)
Constrained Application Protocol, Confirmable, PUT, MID:7140
  01... .... = Version: 1
  ..00 .... = Type: Confirmable (0)
  .... 0000 = Token Length: 0
  Code: PUT (3)
  Message ID: 7140
  Opt Name: #1: Uri-Path: rd
    Opt Desc: Type 11, Critical, Unsafe
    1011 .... = Opt Delta: 11
    .... 0010 = Opt Length: 2
    Uri-Path: rd
  Opt Name: #2: Uri-Path: j5DH4jpBKF
    Opt Desc: Type 11, Critical, Unsafe
    0000 .... = Opt Delta: 0
    .... 1010 = Opt Length: 10
    Uri-Path: j5DH4jpBKF
0000  60 00 00 00 00 00 1a 11 3f aa aa 00 00 00 00 00 00  .....? .....
0010  02 03 53 c3 7d 78 8e b4 aa aa 00 00 00 00 00 00  ..5..x.. .....
0020  00 00 00 00 00 00 00 01 16 34 16 33 00 1a 0b 35  .....4;3...5
0030  40 03 1b e4 b2 72 64 0a 6a 35 44 48 34 6a 70 42  @....rd. j5DH4jpB
0040  4b 46                                     KF

```

Figure 5.31: Registration message sent from the 4LD LWM2M Client (Node 4) to the Leshan Server. The "j5DH4jpBKF" is the registration ID.

It is important to mention that the server timeout in the LWM2M Server object defined in both clients can be modified to another desired value. After the registration, the Web interface shows the list of connected clients:



The screenshot shows the Leshan Server web interface in a Mozilla Firefox browser. The address bar shows 'localhost:8080/#/clients'. The interface has a header with the Leshan logo and navigation tabs for 'Clients' and 'Security'. Below the header, it indicates 'Connected clients: 2'. A table lists the connected clients with columns for 'Client Endpoint', 'Registration ID', and 'Registration Date'.

Client Endpoint	Registration ID	Registration Date
LED DRIVER - NODE 3	FcsVmcuU0z	May 26, 2015 4:56:22 PM
LED DRIVER - NODE 4	88HO0nbVkl	May 26, 2015 6:43:30 PM

Figure 5.32: List of the connected clients in the Leshan Server.

In the Leshan user interface we should click in the name of the client desired to interact with the LWM2M objects and respective resources. For example, by clicking in the "LED DRIVER - NODE 3", the following window will open with all the objects defined in this client:

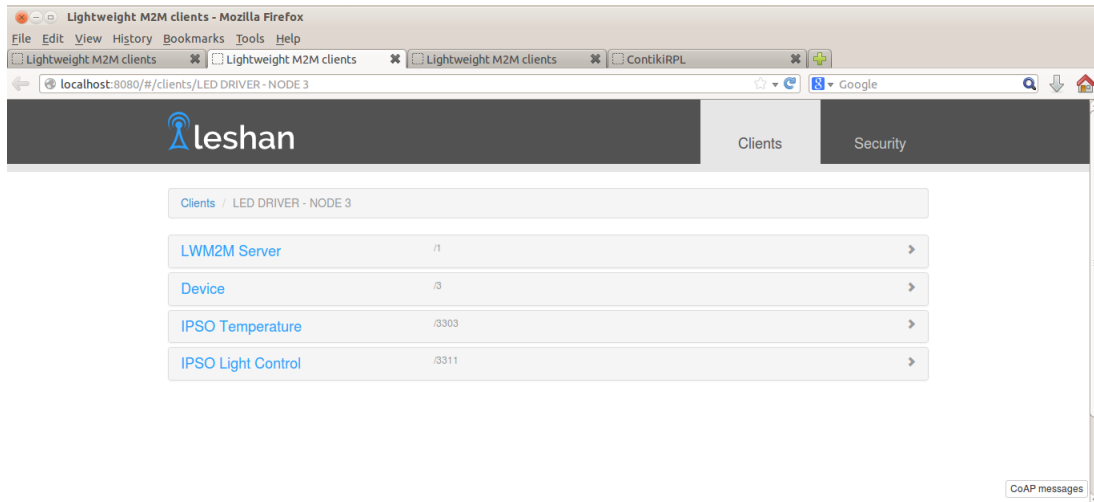


Figure 5.33: List of objects defined in the 4LD Node 3.

To interact with the resources of each object we only need to click in the name of the object desired. For example, by clicking in the "IPSO Light Control" we can see all resources defined in this object:

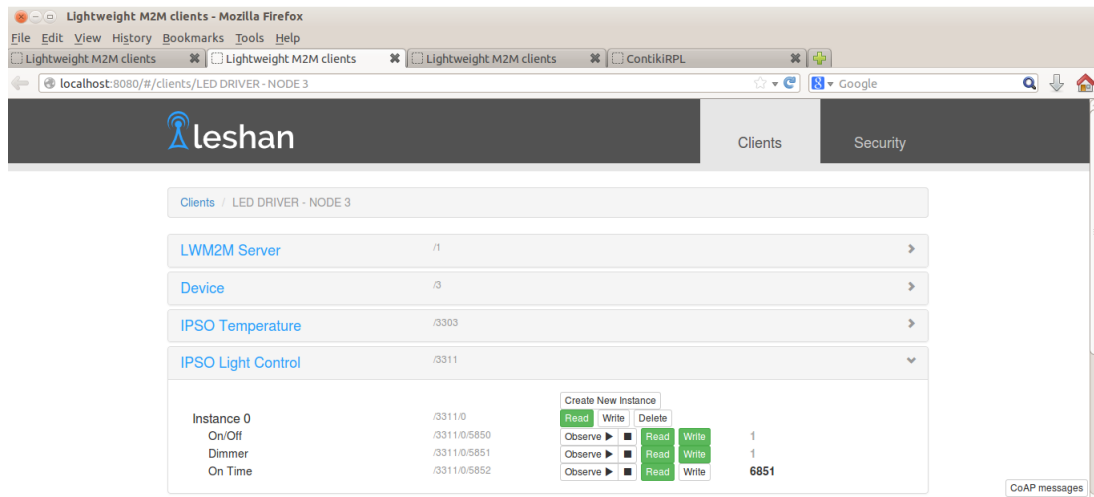


Figure 5.34: List of resources defined in the IPSO Light Control object.

If we read the On/Off resource it is possible to check if the light is on or off. If we write "1" to this resource, the light will be turned on. Otherwise if we write "0" it will be turned off. By reading the Dimmer resource it is possible to obtain the actual PWM duty cycle applied to the light. We can modify this duty cycle by writing values from "0" to "100" on this resource. The On Time resource retrieves the time (in seconds) that the light has been on. We can reset this counter, writing "0" on this resource.

Besides the "IPSO Light Control", if we click in the "IPSO Temperature" it is shown:

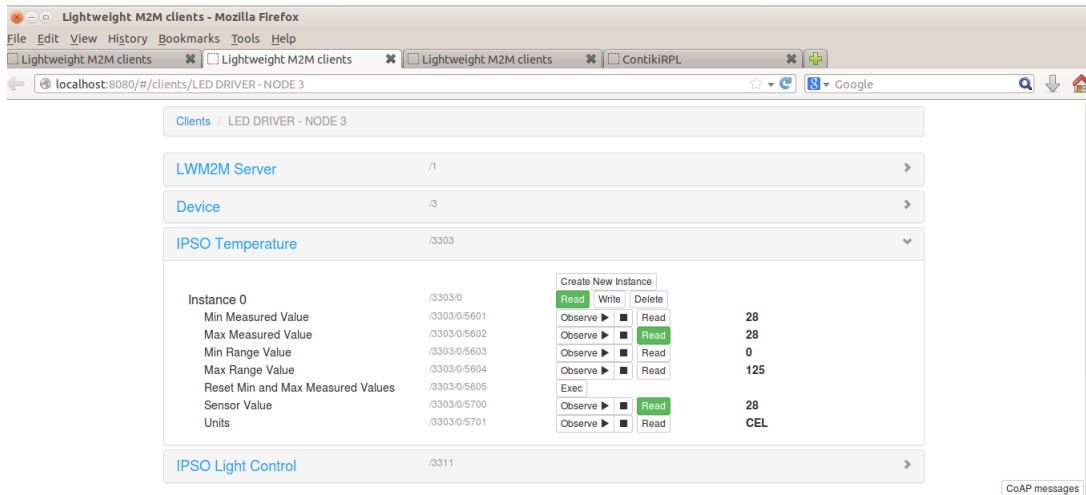


Figure 5.35: List of resources defined in the IPSO Temperature object.

The resources defined in this object and their functionality were already introduced in Chapter 2. Besides these two IPSO objects, we can interact with two additional objects in each client: the LWM2M Server and the Device objects. These LWM2M standard objects are shown in Figures 5.36 and 5.37.

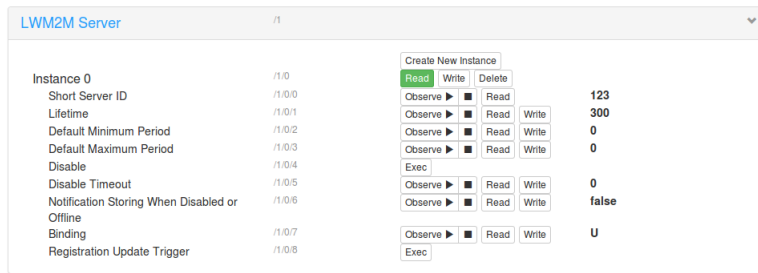


Figure 5.36: List of resources defined in the LWM2M Server object.

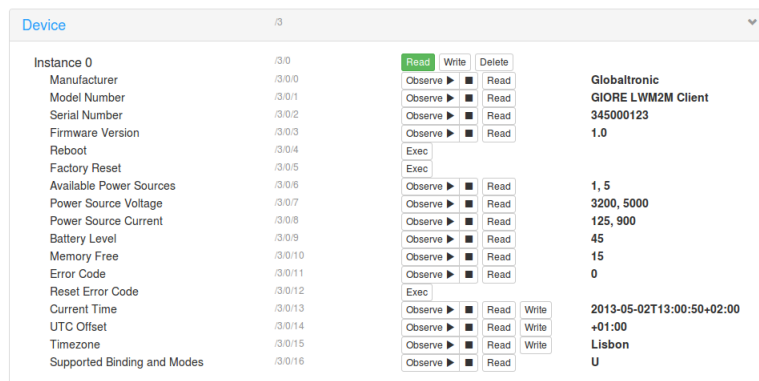


Figure 5.37: List of resources defined in the Device object.

For each resource it is possible to require a notification message each time the resource value changes - CoAP observation feature. This feature can be enabled in the "Observe" button in the

respective resource. When we press the button to stop the observation, the server will send a RST message (empty message) to the client.

5.7 DEVELOPMENT TOOLS

As previously stated in the porting of Contiki OS to the supported hardware platforms and in the two setup experiments, we used the MPLAB® C compiler XC16 for the 4LD platform and the MPLAB® C compiler XC32 for the Giore platform. During the operating system implementation for these new platforms we created four new Makefiles: *Makefile.giore*, *Makefile.leddriver*, *Makefile.pic32* and *Makefile.pic24*, which were already explained in Section 5.3.

The build of the operating system and applications developed was conducted through a UNIX terminal available in InstantContiki virtual machine. To compile one application for a specific platform it is only required to use the following command: *make TARGET=platformname*. For the Giore we need to use *make TARGET=giore*, and for the 4LD platform we use *make TARGET=leddriver*. This command will generate one hex file with extension *.platformname*. This hex file can then be programmed into the platform using the MPLAB IDE and MPLAB ICD3 tool.

Wireshark is a very popular network analyzer tool used during the entire process of development and evaluation. It is perhaps one of the best open source packet analyzers tools that are available nowadays. It allows the user to capture and browse running traffic on the network and it is widely used in educational institutions and also in the industry. Recent implementations include IEEE 802.15.4, 6LoWPAN and CoAP packets dissectors. Other debugging software extensively used for network debugging is ping6 and traceroute6.

5.8 DIFFICULTIES DURING IMPLEMENTATION

The implementation of several elements addressed in this thesis were not straightforward. In this section the main problems that emerged during the development and the best ways to deal with them are discussed.

5.8.1 STACK ISSUES

The lack of sufficient stack space has caused a lot of trouble during the whole project development, specially in the 4LD board. The reason is that this platform only has 8KB of RAM, and in both implementations at least 70% of the memory was needed to store program data, leaving only 2kB to the stack. This limited stack space caused very frequent stack overflows, which resulted in crashes and unexpected behaviour. However, to avoid these problems, stack usage optimisations can be made. Using these optimisations it is possible to reduce the stack usage.

Basically if a function call is made, memory is allocated at the stack. Later, this memory is freed when the function returns. This allocated memory depends on the memory position of the function within the ROM, the number of arguments the functions has and the size of their data types, the

number of local variables in the functions and their sizes. Taking into account these factors, several methods can be used to reduce the stack usage, such as:

1. Avoid the use of function calls - instead it is better to define the function as inline. Using this method we avoid a function call, thus the stack usage is reduced. However, declaring a function as inline can also increase the total RAM and ROM usage if not used with care. This method should be used with precaution and preferably should only be applied for nested function calls. The ROM footprint will eventually increase if an inline function is called from more than one location in the code;
2. Use few function arguments - reducing the number of arguments passed to a function will save some stack space, especially if the arguments have large data types. Whenever an argument is passed along several nested function calls, it might be better to declare this variable as global, instead of passing it to each new function call;
3. Use few local variables - it is possible to allocate the variables with automatic storage duration (on the stack) or with static storage duration (in RAM). Therefore, if we declare local variables as static, stack usage can be reduced. However, we cannot store all local variables into RAM, so we have to define which ones to store in RAM, and which ones to store on the stack. Functions' variables that are reentrant cannot be moved to RAM. In the other hand, variables that are used in frequent called functions and also large variables, should preferably be stored in RAM;
4. Do not use larger data types - this method is applicable for all parts of the code. Memory resources can be efficiently used, if we don't use larger data types than necessary.

Some of the optimisations and improvements must be used with caution, since they may reduce stack usage but at the same time an increase of the remaining memory usage could happen. However, after applying these methods, the code can become harder to understand and maintain.

EVALUATION OF THE IMPLEMENTATION

In order to evaluate the performance of the two implementations and to analyze the capabilities of the system, several tests have been performed. The tests can be classified into three groups: firstly we have verified the memory cost of the two setups implemented, secondly we have analysed the network performance in terms of round trip time and packet loss using the `ping6` command and, finally, we evaluated the response time for the different CoAP resources implemented on the 4LD nodes. This last test was conducted in order to evaluate the implemented CoAP communication stack.

6.1 MEMORY USAGE

The memory usage is a crucial aspect of the implementation, thus it should be analyzed. We want to measure the memory cost of the Internet of Things applications. The results obtained in this section can be used to compare with other implementations, but also to specify memory requirements when designing a new product. There are mainly three different memory types that should be analyzed: RAM, ROM and stack. The stack is actually a reserved part of RAM. In the rest of this section we will refer to RAM as the area of RAM excluding the reserved stack area.

Figure 6.1 shows the Flash and RAM required by the firmware in the first experimental setup. This setup was already presented in Chapter 5. An architecture overview of this setup can be seen in Figure 5.18.

	FLASH (bytes)	FLASH (%)	DATA RAM (bytes)	DATA RAM (%)	STACK (bytes)	STACK (%)
Giore Border Router	84456	16	10656	8	120360	92
4LD CoAP Server Node 3	69570	52	5706	69	2486	31
4LD CoAP Server Node 4	67839	51	5668	69	2524	31

Figure 6.1: Firmware size comparison of the experimental setup using Erbium-CoAP and Copper.

Figure 6.2 shows the Flash and RAM required by the firmware in the second experimental setup. This setup was already presented in Chapter 5. An architecture overview of this setup can be seen in Figure 5.26. It's important to mention that, as said before, this application uses dynamic memory, so we reserved 1024 bytes for the heap.

	FLASH (bytes)	FLASH (%)	DATA RAM (bytes)	DATA RAM (%)	STACK (bytes)	STACK (%)
Giore Border Router	84456	16	10656	8	120360	92
4LD LWM2M Client Node 3	99576	75	6066	74	2126	26
4LD LWM2M Client Node 4	97290	74	5990	73	2202	27

Figure 6.2: Firmware size comparison of the experimental setup using OMA LWM2M and Leshan.

We have tried to break the memory usage down to as many components as possible. For example we tried to understand the capabilities of the 4LD board in terms of memory and related CPU usage. In particular, in order to get the two applications running on the 4LD nodes we had to reduce the routing table for a maximum of 5 neighbors and 5 routes and we also had to reduce the maximum packets that can be holded by the MAC layer. Otherwise, the applications developed would not fit in the 4LD memory resources.

The results of the RAM and stack usage are very similar in both applications. In terms of ROM usage we can see an increase in ROM usage of 30 006 bytes from the first to the second setup.

6.2 NETWORK: PERFORMANCE EVALUATION

The target application of the presented architecture is non-critical services. To evaluate if the system is capable to deliver an adequate quality of service level, we analyzed the network response in terms of Round Trip Time (RTT) and Packet Loss (PLOSS) metrics.

1. Round Trip Time: is the time it takes for a signal to be sent plus the time it takes for an acknowledgment of that signal to be received. In the field of computer networks, the signal is generally a data packet, and the RTT is also known as the ping time;

2. Packet Loss: occurs when one or more data packets traveling across a computer network fail to reach their destination.

Using the experimental setups represented in Figures 5.18 and 5.26, we evaluated the RTT and PLOSS between the Base Station and the 4LD Node making 1000 ping requests for each variation of the IPv6 packet payload. For this purpose we used the ping6 command that uses the Internet Control Message Protocol 6 (ICMPv6).

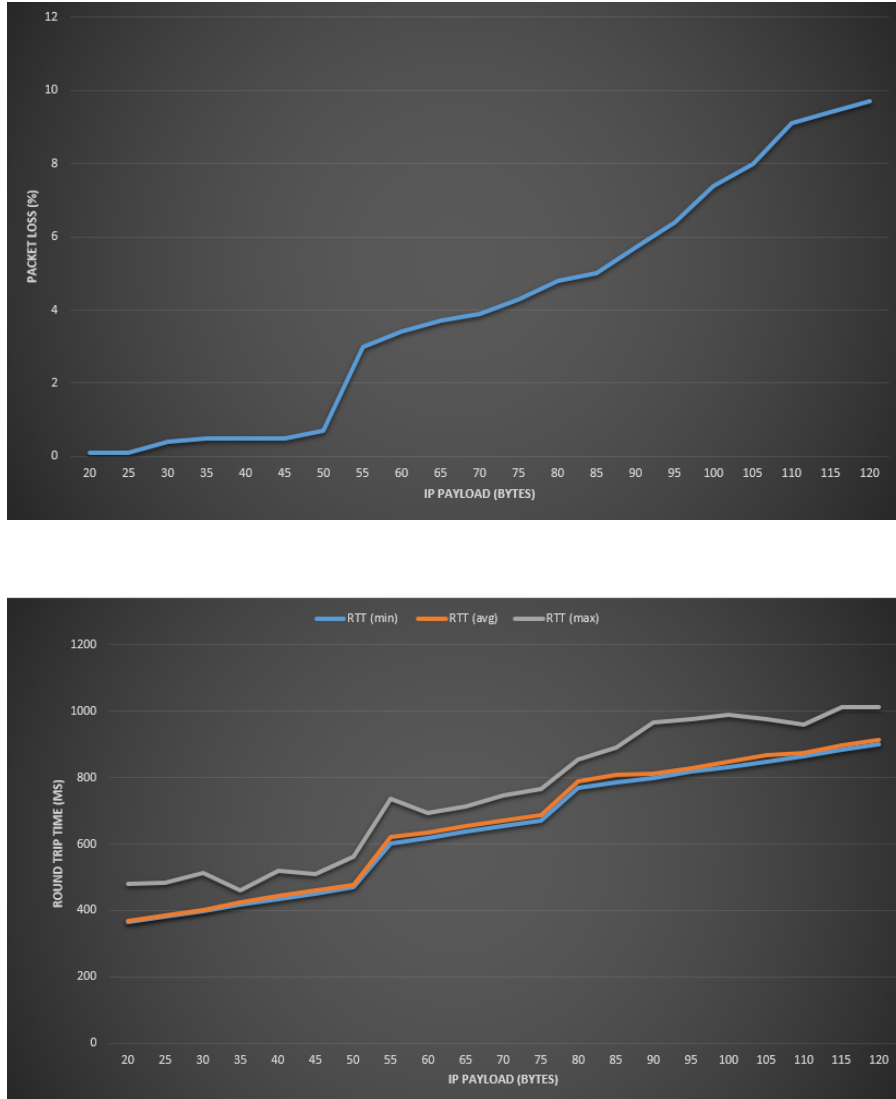


Figure 6.3: RTT and PLOSS evolution according to ICMP payload size for the experimental setup using Erbium-CoAP and Copper.

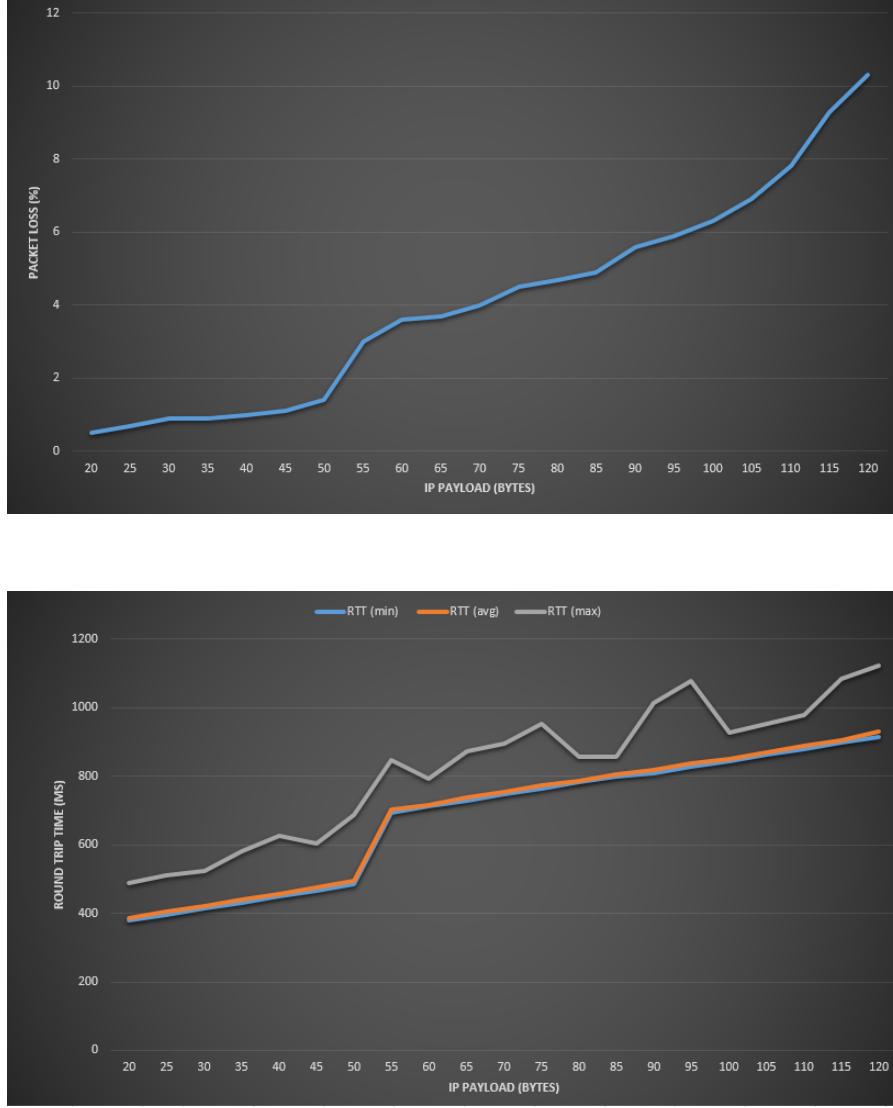


Figure 6.4: RTT and PLOSS evolution according to ICMP payload size for the experimental setup using OMA LWM2M and Leshan.

Figures 6.3 and 6.4 show an increasing packet loss starting from 55 bytes of ICMP payload in both setups implemented, which corresponds to a packet with a total of 103 bytes, including the 48 bytes of the IPv6 header. The maximum packet size for IEEE 802.15.4 is 127 bytes and IPv6 requires that the underlying layers support packets of at least 1280 bytes. This means that the link layer fragmentation support must exist. Taking into account that the IEEE 802.15.4 header has at least 25 bytes and the IEEE 802.15.4 maximum packet size is 127 bytes, we only can send a maximum of 102 bytes of data in one packet. So, if a packet payload has more than 102 bytes, the packet must be fragmented. Hence, we can conclude that the packet loss increase after the 55 bytes of ICMP payload is caused by the IP fragmentation mechanism.

Another characteristic that can be responsible for the packet loss increase is related to limited capacity for buffering packets at physical and MAC layers, since the transceiver resources and main memory can accommodate few packets at a time. In the case of the 4LD CoAP server the MAC layer can hold only 3 packets.

If we look at the round trip time graphics, we can see that the RTT gradually increases as the payload increases. This was expected since the payload data to process by the CoAP server is increasing. We can also check some peaks in the maximum RTT, that we attributed to the fact that we are facing a system triggered by timers. If, when sending the ICMP request, the platform is busy processing other information, the response may take longer to be received. We can also see that there are areas where the RTT increases faster. This is due to the packet fragmentation. For example at 55 bytes of ICMP payload, the packet is sent using two fragments, so it is clear that the processing will take more time.

Despite the packet loss issue, solved by the CoAP protocol through Blockwise Transfers pattern, the graphics clearly demonstrate that the implemented 6LoWPAN networking stack is working and that the experimental values of RTT and PLOSS are small enough to allow non-critical applications.

6.3 COAP TRANSACTIONS: PERFORMANCE EVALUATION

This section presents some tests that have been made in order to evaluate the implemented CoAP communication stack. The parameter that has been measured is the response time, which describes how fast a CoAP transaction is completed for the different resources.

Figures 6.5 and 6.7 show a few results taken to evaluate the performance when retrieving different resources on the CoAP server, for both setups implemented (5.18 and 5.26). The retrieval time is measured on the base station to show the time taken to retrieve a given resource. Figures 6.6 and 6.8 shows the total number of packets needed to retrieve all the information from a specific resource and the number of bytes transmitted (requests + acknowledges).

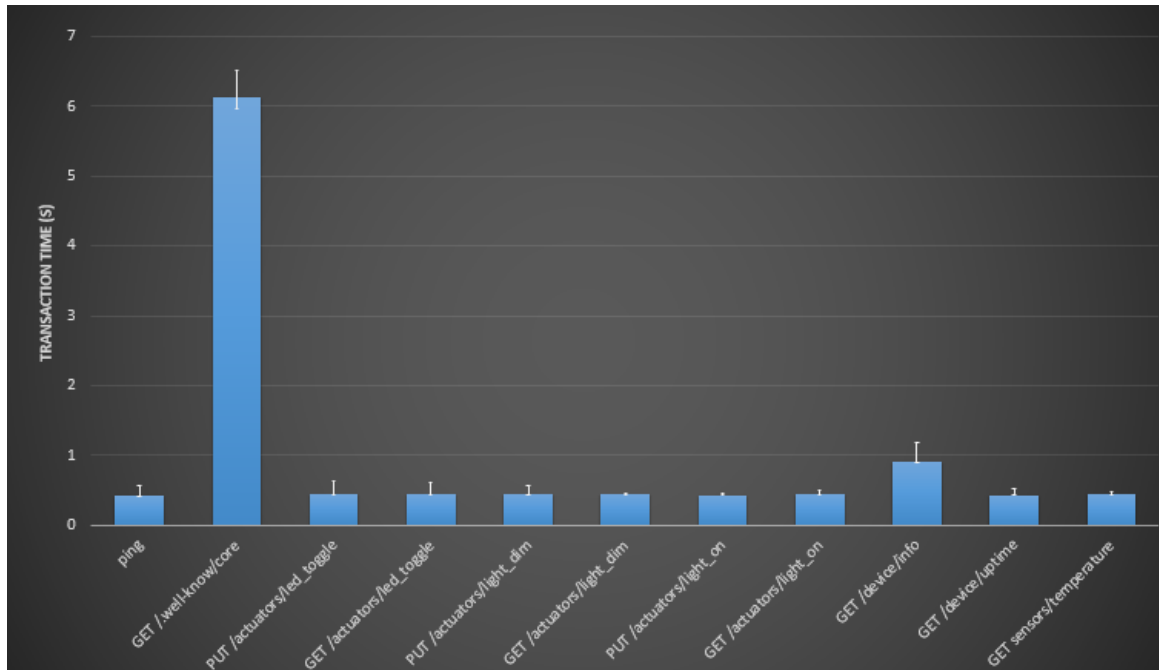


Figure 6.5: Response time of the CoAP resource requests for the experimental setup using Erbium-CoAP and Copper. The response time shown is the average result of 100 samples.

	GET /.well-known/core	PUT /actuators/led_toggle	GET /actuators/led_toggle	PUT /actuators/light_dim	GET /actuators/light_dim
Number of packets transmitted	24	2	2	2	2
Number of bytes transmitted	1892	127	144	126	145

	PUT /actuators/light_on	GET /actuators/light_on	GET /device/info	GET /device/uptime	GET /sensors/temperature
Number of packets transmitted	2	2	4	2	2
Number of bytes transmitted	125	144	295	143	149

Figure 6.6: Total number of bytes needed to retrieve all the information from each resource. These results are for the experimental setup using Erbium-CoAP and Copper.

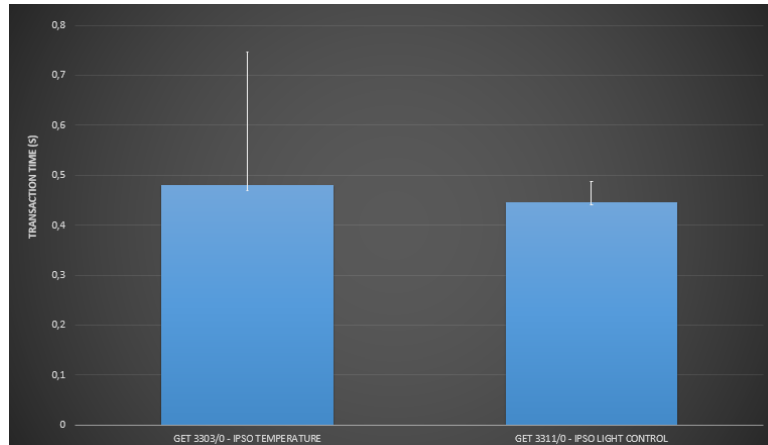


Figure 6.7: Response time of the CoAP resource requests for the experimental setup using OMA LWM2M and Leshan. The response time shown is the average result of 100 samples.

	GET 3303/0	GET /3311/0
Number of packets transmitted	2	2
Number of bytes transmitted	132	146

Figure 6.8: Total number of bytes needed to retrieve all the information from each resource. These results are for the experimental setup using OMA LWM2M and Leshan.

From these figures it is clear that GET */.well-known/core* is the resource request that takes more time. This was expected since it is the resource that has the highest number of bytes to be transmitted. It is important to mention that for Figure 6.7, instead of individual resources the retrieval time for two IPSO Objects (IPSO Temperature and IPSO Light Control) is shown. As documented in Figure 2.13, the IPSO Temperature object has 7 resources implemented and the IPSO Light Control has 3

resources implemented. The retrieval time presented in this graphic is the time needed to retrieve the data from all resources available in each object. From the collected data we can conclude that the implementation using OMA LWM2M is more efficient, since it can retrieve the information from 7 different resources with only 2 packets and a total of 132 bytes transmitted. This is achieved using the TLV (Type-Length-Value) message encoding supported by the Leshan Server.

CONCLUSIONS

This chapter summarizes the contributions of this project providing some conclusions to take into consideration. Afterwards, some ideas for future enhancements and upgrades are presented.

7.1 CONCLUSIONS

This thesis presented and evaluated an architecture that supports WSNs applications in the field of Intelligent Street Lighting. This architecture was built using a set of promising protocols. Based on the results presented in Chapter 6, we can state that it is possible to deploy an Intelligent Street Lighting network and develop complex applications on top of it.

The preliminary results obtained are rather satisfying. The Contiki port to the PIC24FJ microcontroller along with a 6LoWPAN stack has been implemented and tested. The analysis shows that 6LoWPAN and the RPL software available on Contiki, and also the remaining features available, matches the application requirements in a vast spectrum of application domains.

The use of CoAP together with 6LoWPAN performs satisfactorily. It significantly reduces the size of the packets sent, which is important for wireless transmissions in resource constrained devices. An HTTP and TCP implementation would not have been possible to fit in our devices, especially in the 4LD board.

Using the OMA LWM2M objects on top of the CoAP implementation proved to be very efficient. CoAP and OMA LWM2M protocols are considered the key components of the future global standardized M2M architecture. The results found in this work show that the interoperability provided by the LWM2M protocol is excellent and is showing good maturity on a basic level.

Choosing Contiki for implementing our WSN has significantly simplified our work, since it already includes support for most of the protocols required in each layer. It would not be reasonable to write a solution starting from scratch, since this would require a significant amount of time and would be less stable than more mature and well-tested software. In this work we have tried to use as many pre-written pieces of software as possible, and focused on integrating them on Contiki. Using open-source software

was a requirement, since we wanted to be able to adapt the software to our specific needs without additional costs. Besides, Contiki is a well-supported OS and there is a lot of activity on its mailing list and on their code repository. This is crucial because it shows that the project is actively developed, used and tested.

We can claim that, at least for non-critical applications, the presented architecture can compete with other solutions based on expensive and general purpose technologies. This architecture can be readily deployed and interconnected with a legacy infrastructure. Moreover, due to the scalability and versatility of the protocols used and the adoption of an IPv6 addressing for the sensor nodes, the presented architecture is easy to be maintained and upgraded in order to support additional functionalities.

7.2 FUTURE WORK

There is a significant amount of work that could be conducted in possible future implementations. Using our current hardware, there are some improvements that can be made:

1. Try to optimise the stack usage even further. It should be noted that this probably requires widespread changes in code, with patches that are unsuitable for other platforms;
2. Add a SRAM to the actual PIC24F so we can extend the data memory space. Through this we can extend the stack size and also be able to support more CoAP resources;
3. Try to implement the LWM2M application using only static memory, so we can prevent memory fragmentation problems and solve the unexpected crashes in the long stability tests. We can also try to use a deterministic dynamic memory manager;
4. Add support in the current implementation for upgrading the firmware over-the-air, since it would be very useful when deploying this type of sensor networks on the field;
5. Use a different RDC driver, for example, the ContikiRPL. Provided that in our implementation the radio transceiver is always on (*nullrdc* driver), the energy is not spent efficiently. Using the ContikiRPL the radio is kept off as much as possible.

Another option is to use different hardware, presumably hardware that has more resources available, especially more stack memory and processing power. Instead of using the 4LD with the PIC24F microprocessor as CoAP server, we can use the Giore that has the PIC32MX. We already ported Contiki to this platform and because of the increase of stack, RAM and processing power, we believe that it would solve several issues with our current implementation.

REFERENCES

- [1] K. S. Low, W. Win, and M. J. Er, “Wireless sensor networks for industrial environments”, in *Computational Intelligence for Modelling, Control and Automation, 2005 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on*, vol. 2, Nov. 2005, pp. 271–276. DOI: 10.1109/CIMCA.2005.1631480.
- [2] V. Gungor and G. Hancke, “Industrial wireless sensor networks: Challenges, design principles, and technical approaches”, *Industrial Electronics, IEEE Transactions on*, vol. 56, no. 10, pp. 4258–4265, Oct. 2009, ISSN: 0278-0046. DOI: 10.1109/TIE.2009.2015754.
- [3] T. Arampatzis, J. Lygeros, and S. Manesis, “A survey of applications of wireless sensors and wireless sensor networks”, in *Intelligent Control, 2005. Proceedings of the 2005 IEEE International Symposium on, Mediterrean Conference on Control and Automation*, Jun. 2005, pp. 719–724.
- [4] V. Potdar, A. Sharif, and E. Chang, “Wireless sensor networks: A survey”, in *Advanced Information Networking and Applications Workshops, 2009. WAINA '09. International Conference on*, May 2009, pp. 636–641. DOI: 10.1109/WAINA.2009.192.
- [5] N. Xu, “A survey of sensor network applications”, *IEEE Communications Magazine*, vol. 40, 2002.
- [6] Z. Shelby, K. Hartke, C. Bormann, and B. Frank, “Constrained application protocol (coap)”, IETF Secretariat, Fremont, CA, USA, Tech. Rep. draft-ietf-core-coap-13.txt, Dec. 6, 2012. [Online]. Available: <https://tools.ietf.org/html/draft-ietf-core-coap-13>.
- [7] J.-P. Vasseur and A. Dunkels, *Interconnecting Smart Objects with IP: The Next Internet*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010, ISBN: 0123751659, 9780123751652.
- [8] P. Sausen, A. Sausen, F. Salvadori, R. E. Júnior, and M. de Campos, *Development and implementation of wireless sensor network for the electricity substation monitoring*, 2012.
- [9] S. Kosmerchock, *Wireless sensor network topologies*. [Online]. Available: k5systems.com/TP0001_v1.pdf.
- [10] J. Gutierrez, M. Naeve, E. Callaway, M. Bourgeois, V. Mitter, and B. Heile, “Ieee 802.15.4: A developing standard for low-power low-cost wireless personal area networks”, *Network, IEEE*, vol. 15, no. 5, pp. 12–19, Sep. 2001, ISSN: 0890-8044. DOI: 10.1109/65.953229.
- [11] P. Baronti, P. Pillai, V. W. Chook, S. Chessa, A. Gotta, and Y. F. Hu, “Wireless sensor networks: A survey on the state of the art and the 802.15.4 and zigbee standards”, *Computer Communications*, vol. 30, no. 7, pp. 1655–1695, 2007, Wired/Wireless Internet Communications, ISSN: 0140-3664. DOI: <http://dx.doi.org/10.1016/j.comcom.2006.12.020>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0140366406004749>.
- [12] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th. Wiley Publishing, 2008, ISBN: 0470128720.

- [13] A. Mallikarjuna, R. V. P. Kumar, D. Janakiram, and G. A. Kumar, “Operating systems for wireless sensor networks: a survey technical report”,
- [14] J. Ousterhout, “Why threads are a bad idea (for most purposes)”, in *USENIX Winter Technical Conference*, Jan. 1996. [Online]. Available: <http://www.cs.utah.edu/~regehr/research/ouster.pdf>,%20http://home.pacbell.net/ouster/threads.ppt.
- [15] R. von Behren, J. Condit, and E. Brewer, “Why events are a bad idea (for high-concurrency servers)”, in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, ser. HOTOS’03, Lihue, Hawaii: USENIX Association, 2003, pp. 4–4. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1251054.1251058>.
- [16] G. C. Buttazzo, “Rate monotonic vs. edf: Judgment day”, *Real-Time Syst.*, vol. 29, no. 1, pp. 5–26, Jan. 2005, ISSN: 0922-6443. DOI: 10.1023/B:TIME.0000048932.30002.d9. [Online]. Available: <http://dx.doi.org/10.1023/B:TIME.0000048932.30002.d9>.
- [17] N. Kushalnagar, G. Montenegro, and C. P. Schumacher, “IPv6 over low-power wireless personal area networks (6LoWPANs): Overview, assumptions, problem statement, and goals”, RFC Editor, Fremont, CA, USA, RFC 4919, Aug. 2007. [Online]. Available: <http://www.rfc-editor.org/rfc/rfc4919.txt>.
- [18] N. Kushalnagar, G. Montenegro, D. E. Culler, and J. W. Hui, “Transmission of ipv6 packets over ieee 802.15.4 networks”, RFC Editor, Fremont, CA, USA, Tech. Rep. 4944, Sep. 2007. [Online]. Available: <http://www.rfc-editor.org/rfc/rfc4944.txt>.
- [19] Z. Shelby and C. Bormann, *6LoWPAN: The Wireless Embedded Internet*. Wiley Publishing, 2010, ISBN: 0470747994, 9780470747995.
- [20] E. J. Hui and P. Thubert, “Compression format for ipv6 datagrams over ieee 802.15.4-based networks”, RFC Editor, Tech. Rep. 6282, Sep. 2011. [Online]. Available: <https://tools.ietf.org/html/rfc6282>.
- [21] D. C. J. Hui and S. Chakrabarti, “6lowpan: Incorporating IEEE 802.15.4 into the IP architecture, IPSO alliance white paper”, 2009.
- [22] E. Z. Shelby, S. Chakrabarti, E. Nordmark, and C. Bormann, “Neighbor discovery optimization for ipv6 over low-power wireless personal area networks (6lowpans)”, RFC Editor, Tech. Rep. 6775, Nov. 2012. [Online]. Available: <https://tools.ietf.org/html/rfc6775>.
- [23] A. H. Chowdhury, M. Ikram, H.-S. Cha, H. Redwan, S. M. S. Shams, K.-H. Kim, and S.-W. Yoo, “Route-over vs mesh-under routing in 6lowpan”, in *Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing: Connecting the World Wirelessly*, ser. IWCMC ’09, Leipzig, Germany: ACM, 2009, pp. 1208–1212, ISBN: 978-1-60558-569-7. DOI: 10.1145/1582379.1582643. [Online]. Available: <http://doi.acm.org/10.1145/1582379.1582643>.
- [24] T. Winter, P. Thubert, 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*, RFC 6550 (Proposed Standard), Mar. 2012. [Online]. Available: <http://www.ietf.org/rfc/rfc6550.txt>.
- [25] J. Vasseur, N. Agarwal, J. Hui, Z. Shelby, P. Bertrand, and C. Chauvenet, “Rpl: The ip routing protocol designed for low power and lossy networks, IPSO alliance white paper”, 2011.
- [26] W. Colitti, K. Steenhaut, and N. D. Caro, “De integrating wireless sensor networks with the web”, in *In Proceedings of Workshop on Extending the Internet to Low power and Lossy Networks*, 2011.
- [27] C. Bormann, A. P. Castellani, and Z. Shelby, “Coap: An application protocol for billions of tiny internet nodes”, *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, 2012, ISSN: 1089-7801. DOI: <http://doi.ieeecomputersociety.org/10.1109/MIC.2012.29>.

- [28] E. Z. Shelby, “Block-wise transfers in coap”, Tech. Rep., Mar. 2015. [Online]. Available: <https://www.ietf.org/id/draft-ietf-core-block-17.txt>.
- [29] M. Castro, A. J. Jara, and A. Skarmeta, “Architecture for improving terrestrial logistics based on the web of things”, *Sensors*, vol. 12, no. 5, p. 6538, 2012, ISSN: 1424-8220. DOI: 10.3390/s120506538. [Online]. Available: <http://www.mdpi.com/1424-8220/12/5/6538>.
- [30] K. Hartke, “Observing resources in coap”, IETF Secretariat, Fremont, CA, USA, Tech. Rep. draft-ietf-core-observe-08.txt, Feb. 25, 2013. [Online]. Available: <http://www.rfc-editor.org/internet-drafts/draft-ietf-core-observe-08.txt>.
- [31] M. Nottingham and E. Hammer-Lahav, “Defining well-known uniform resource identifiers (uris), rfc 5785 (proposed standard)”, 2010.
- [32] Z. Shelby, “Constrained restful environments (core) link format, rfc 6690 (proposed standard)”, 2012.
- [33] G. Klas, F. Rodermund, Z. Shelby, S. Akhour, and J. Höller, *Lightweight m2m: enabling device management and applications for the internet of things*, 2014. [Online]. Available: <http://archive.ericsson.net/service/internet/picov/get?DocNo=1/28701-FGB101973>.
- [34] *Open mobile alliance - lightweight m2m v1.0 technical information*. [Online]. Available: <http://technical.openmobilealliance.org/Technical/technical-information/release-program/current-releases/oma-lightweightm2m-v1-0>.
- [35] *Open mobile alliance - lightweight m2m object and resource registry*. [Online]. Available: <http://technical.openmobilealliance.org/Technical/technical-information/omna/lightweight-m2m-lwm2m-object-registry>.
- [36] K. Romer and F. Mattern, “The design space of wireless sensor networks”, *Wireless Communications, IEEE*, vol. 11, no. 6, pp. 54–61, Dec. 2004, ISSN: 1536-1284. DOI: 10.1109/MWC.2004.1368897.
- [37] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, “System architecture directions for networked sensors”, *SIGARCH Comput. Archit. News*, vol. 28, no. 5, pp. 93–104, Nov. 2000, ISSN: 0163-5964. DOI: 10.1145/378995.379006. [Online]. Available: <http://doi.acm.org/10.1145/378995.379006>.
- [38] T. Stathopoulos, J. Heidemann, and D. Estrin, “A remote code update mechanism for wireless sensor networks”, Tech. Rep., 2003.
- [39] J. W. Hui and D. Culler, “The dynamic behavior of a data dissemination protocol for network programming at scale”, in *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys ’04, Baltimore, MD, USA: ACM, 2004, pp. 81–94, ISBN: 1-58113-879-2. DOI: 10.1145/1031495.1031506. [Online]. Available: <http://doi.acm.org/10.1145/1031495.1031506>.
- [40] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, “Tinyos: An operating system for sensor networks”, in *In Ambient Intelligence*, Springer Verlag, 2004.
- [41] M. O. Farooq and T. Kunz, “Operating systems for wireless sensor networks: A survey”, *Sensors*, vol. 11, no. 6, pp. 5900–5930, 2011, ISSN: 1424-8220. DOI: 10.3390/s110605900. [Online]. Available: <http://www.mdpi.com/1424-8220/11/6/5900>.
- [42] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesc language: A holistic approach to networked embedded systems”, in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI ’03, San Diego, California, USA: ACM, 2003, pp. 1–11, ISBN: 1-58113-662-5. DOI: 10.1145/781131.781133. [Online]. Available: <http://doi.acm.org/10.1145/781131.781133>.

- [43] *Network protocols tinys wiki - dissemination*. [Online]. Available: <http://tinys.stanford.edu/tinys-wiki/index.php/Dissemination>.
- [44] *Network protocols tinys wiki - tymo*. [Online]. Available: <http://tinys.stanford.edu/tinys-wiki/index.php/Tymo>.
- [45] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors", in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, ser. LCN '04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462, ISBN: 0-7695-2260-2. DOI: 10.1109/LCN.2004.38. [Online]. Available: <http://dx.doi.org/10.1109/LCN.2004.38>.
- [46] A. K. Dwivedi, M. K. Tiwari, and O. P. Vyas, "Operating systems for tiny networked sensors: A survey", *IJRTE 2009*, pp. 152–157,
- [47] A. Dunkels and O. Schmidt, *Protothreads - lightweight, stackless threads in c*, 2005.
- [48] *Contiki documentation*. [Online]. Available: <http://contiki.sourceforge.net/docs/2.6/>.
- [49] N. Tsiftes, J. Eriksson, and A. Dunkels, "Low-power wireless ipv6 routing with contiki-rl", in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, ser. IPSN '10, Stockholm, Sweden: ACM, 2010, pp. 406–407, ISBN: 978-1-60558-988-6. DOI: 10.1145/1791212.1791277. [Online]. Available: <http://doi.acm.org/10.1145/1791212.1791277>.
- [50] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The liteos operating system: Towards unix-like abstractions for wireless sensor networks", in *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, Apr. 2008, pp. 233–244. DOI: 10.1109/IPSN.2008.54.
- [51] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-rk: An energy-aware resource-centric rtos for sensor networks", in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, Dec. 2005, pages. DOI: 10.1109/RTSS.2005.30.
- [52] A. Rowe, K. Lakshmanan, H. Zhu, and R. Rajkumar, "Rate-harmonized scheduling for saving energy", in *Proceedings of the 2008 Real-Time Systems Symposium*, ser. RTSS '08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 113–122, ISBN: 978-0-7695-3477-0. DOI: 10.1109/RTSS.2008.50. [Online]. Available: <http://dx.doi.org/10.1109/RTSS.2008.50>.
- [53] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms", *Mob. Netw. Appl.*, vol. 10, no. 4, pp. 563–579, Aug. 2005, ISSN: 1383-469X. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1160162.1160178>.
- [54] A. Dunkels, "Programming memory-constrained networked embedded systems", PhD thesis, Swedish Institute of Computer Science, Stockholm, Sweden, 2007. [Online]. Available: <http://www.sics.se/~adam/dunkels07programming.pdf>.
- [55] A. Dunkels, "Poster abstract rime a lightweight layered communication stack for sensor networks", 2007. [Online]. Available: <http://www.sics.se/~adam/dunkels07rime.pdf>.
- [56] *Contiki os wiki - processes*, 2011. [Online]. Available: <https://github.com/contiki-os/contiki/wiki/Processes>.
- [57] A. Valente, "Implantação de sistemas operativos em módulos de comunicação sem fios", Master's thesis, University of Aveiro, 2013.
- [58] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, *Protothreads: simplifying event-driven programming of memory-constrained embedded systems*, 2006. [Online]. Available: <http://www.sics.se/~adam/dunkels06protothreads.pdf>.

- [59] *Contiki os wiki - multithreading*, 2011. [Online]. Available: <https://github.com/contiki-os/contiki/wiki/Multithreading>.
- [60] *Contiki os wiki - memory allocation*, 2011. [Online]. Available: <https://github.com/contiki-os/contiki/wiki/Memory-allocation>.
- [61] *Contiki os wiki - file systems*, 2011. [Online]. Available: <https://github.com/contiki-os/contiki/wiki/File-systems>.
- [62] *Contiki os wiki - dynamic loader*, 2011. [Online]. Available: <https://github.com/contiki-os/contiki/wiki/The-dynamic-loader>.
- [63] *Contiki os wiki - timers*, 2011. [Online]. Available: <https://github.com/contiki-os/contiki/wiki/Timers>.
- [64] *Contiki os wiki - i/o*, 2011. [Online]. Available: <https://github.com/contiki-os/contiki/wiki/Input-and-output>.
- [65] A. Dunkels, “Towards tcp/ip for wireless sensor networks”, Master’s thesis, Swedish Institute of Computer Science, tockholm, Sweden, 2005. [Online]. Available: <http://www.sics.se/~adam/dunkels05towards.pdf>.
- [66] A. Dunkels, F. Österlind, and Z. He, “An adaptive communication architecture for wireless sensor networks”, in *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, ser. SenSys ’07, Sydney, Australia: ACM, 2007, pp. 335–349, ISBN: 978-1-59593-763-6. DOI: 10.1145/1322263.1322295. [Online]. Available: <http://doi.acm.org/10.1145/1322263.1322295>.
- [67] *Networks of embedded systems group home page*. [Online]. Available: <http://rtn.sssup.it/index.php/software/contiki>.
- [68] *Contiki 2.6 doxygen documentation*. [Online]. Available: <http://contiki.sourceforge.net/docs/2.6>.
- [69] C. Yibo, K.-m. Hou, H. Zhou, H.-L. Shi, X. Liu, X. Diao, H. Ding, J.-J. Li, and C. de Vaulx, “6lowpan stacks: A survey”, in *Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference on*, Sep. 2011, pp. 1–4. DOI: 10.1109/wicom.2011.6040344.
- [70] J. Ko, J. Eriksson, N. Tsiftes, S. Dawson-haggerty, A. Terzis, A. Dunkels, and D. Culler, “Contikirpl and tinyrpl: Happy together”, in *In Proceedings of the workshop on Extending the Internet to Low power and Lossy Networks (IP+SN 2011)*, 2011.