POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione

Master of Science in Computer Engineering

Design Development and Assessment of a Multi-interface IoT Platform

Supervisor: Prof. William Fornaciari

Co-Supervisor: Ing. Stefano Bosisio,
Ing. David Siorpaes,
PhD Giuseppe Massari,

Master Thesis by:

Michele Liscio Matr. 819074

Abstract

ENG

The Internet of Things is the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment [1]. Any object can thus become potentially interactive thanks to the internet.

This master thesis project proposes a network prototype Contiki-based and 6LoWPAN-based that implements a multi-interface border router that can be used in some Internet of Things (IoT) scenarios. This prototype wants to enable new interoperability scenarios among nodes belonging to a WSN, which would be allowed to utilize more than one communication interface. Possible scenarios include home automation and industrial field, where the border router node would collect and route different subnets data traffic and where every subnets would utilize a different communication technology.

This project analyzes the different issues that arises with this network prototype, such the nodes and network configuration and the packet routing.

The proposed solution:

- has been implemented in Contiki, the de-facto standard OS for low-power/memory-constrained IoT devices
- is based on 6LoWPAN protocol, the IETF's standard that introduces IPv6 in the IoT
- uses and integrates the already existing Operating System (OS)'s routing protocols and OS data structures

- it's scalable: it's possible to add as many interfaces as needed to the border router without having to modify the kernel
- it has been tested in a real-case scenario, that is in a network based on two communication technologies: a sub-1 GHz RF radio channel and a power-line
- introduces a low overhead on the application footprint
- has low impact on system general performances

Keywords: 6LoWPAN, Contiki, RPL, Internet of Things, multi-interface, multi-radio, IPv6

ITA

Internet of Things è la rete degli oggetti fisici che contengono tecnologia embedded utilizzata per comunicare, avvertire o interagire con i loro stati interni o con l'ambiente esterno [1]. Tutti gli oggetti dunque possono potenzialmente acquisire un ruolo attivo grazie al collegamento alla Rete.

Questo lavoro di tesi propone un prototipo di rete basato su Contiki e su 6LoWPAN che implementa un nodo border router multi-interfaccia utilizzabile in alcuni scenari del mondo Internet of Things (IoT). Tale prototipo vuole aprire nuovi scenari di interoperabilità tra i nodi di una rete WSN, la quale sarebbe così svincolata dall'avere una sola interfaccia di comunicazione tra nodi. Possibili scenari di utilizzo comprendono la domotica e l'ambito industriale, nei quali il nodo border router raccoglierebbe ed instraderebbe il traffico dati di diverse sottoreti, ognuna basata su una tecnologia di comunicazione differente.

In questa tesi sono state studiate le differenti problematiche afferenti questo prototipo di rete, quali la configurazione dei nodi e i meccanismi di instradamento dei pacchetti.

La soluzione proposta:

- è stata implementata su Contiki, l'OS standard de-facto nel mondo dei dispositivi low-power/memory-constrained dell'IoT
- è basata sul protocollo 6LoWPAN, lo standard IETF che introduce l'utilizzo dell'IPv6 nell'IoT

- sfrutta e integra i protocolli di routing e le strutture dati già esistenti nell'OS
- è scalabile: è possibile aggiungere al border router un numero arbitrario di interfacce senza dover apportare modifiche al kernel
- è stata testata in un caso reale di utilizzo, cioè in una rete che sfrutta due tecnologie di comunicazione: una radio sub-1 GHz RF ed un powerline modem
- introduce un basso overhead sul footprint del programma
- ha basso impatto sulle performance generali del sistema

Parole chiave: 6LoWPAN, Contiki, RPL, Internet of Things, multi-interface, multi-radio, IPv6

Acknowledgements

Il mio riconoscimento più grande va ai miei genitori, senza il quale supporto, emotivo ed economico, il raggiungimento di questo traguardo non sarebbe mai stato possibile.

Desidero poi ringraziare i miei tutor aziendali, Stefano e David, per il loro supporto tecnico e per la loro disponibilità sia in fase di sviluppo del progetto, sia in fase di revisione della tesi.

Desidero infine ringraziare Serena per tutto il supporto e l'amore datomi in questi ultimi tre anni. Sono stati indubbiamente gli anni più belli e formativi della mia vita. Grazie di avermi reso una persona felice, ma soprattutto, grazie di avermi reso una persona migliore. Molto migliore di quanto un riconoscimento cartaceo possa mai fare.

Milano, July 2016 L. M.

Contents

Introduction 1

1 Internet of Things 5

1.1 ITU Enabling Technologies . . . 8
1.2 Near-Field Communication (NFC) . . . 14
1.3 ISM band . . . 15
1.4 LoRa . . . 18
1.5 Wi-Fi . . . 19
1.6 Bluetooth . . . 19
1.7 OSI network stack . . . 21
1.8 Internet Protocol Suite . . . 26
1.9 IEEE 802.15.4 . . . 29
1.10 CoAP . . . 33
1.11 ICMP . . . 34
1.12 IPv6 . . . 34
1.13 6LoWPAN . . . 48
1.14 Neighbor Discovery Protocol for IPv6 . . . 53
1.15 ND over 6LoWPANs . . . 61
1.16 RPL . . . 68
1.17 Power-Line Communication (PLC) . . . 76

1.17.1 Narrowband PLC 76
1.17.2 Broadband PLC 79

2 State of the Art 81

2.1 Hardware 81
2.2 Software 82
2.2.1 Contiki 3.X OS 84
2.2.2 Routing Tables 94
2.3 Other OSs 96

3 Proposed Solution 99

3.1 Hardware Implementation 100
3.2 Software Modifications 112
3.2.1 Platform Porting 112
3.2.2 RPL Subnetting 116
3.2.3 Netstack Vectorization 124
3.2.4 interface Structure Vectorization 131
3.3 Software Working Configurations 143
3.4 Software Implementation 145

4 Evaluation 149

4.1 Memory Footprint 149
4.2 Performance 157
4.2.1 Packet Netstack Crossing Performance 158
4.2.2 Multicast DIO Output Performance 164
4.2.3 BR RPL Instance Initialization Performance 165

Conclusions 169

A Appendix **171**

A.1 Border-Router GPIOs Configuration 171

A.2 ARM Cortex Cycle Counter Handling 173

A.3 Debugging the Alpha Configuration 175

Acronyms **177**

Bibliography **179**

List of Figures

1 Internet of Things trend 2

1.1 source: ITU, adapted from the Nomura Research Institute, "Ubiquitous Networking: Business Opportunities and Strategic Issues", August 2004 6

1.2 OSI model 21

1.3 IPv6 header 37

1.4 IPv6 plus UDP header 50

1.5 6LoWPAN header in its simplest form 51

1.6 IEEE 802.15.4 frame with 6LoWPAN/UDP header 51

1.7 LLN: Mesh VS RouteOver 52

1.8 ND6: RS and RA exchange 67

1.9 ND6: NS and NA exchange, with registration 67

1.10 RPL instance example 72

1.11 RPL DIO message 73

1.12 RPL Control message 74

2.1 Contiki Netstack 89

2.2 Netstack layers interaction 93

2.3 Network composition of the routing tables example 95

2.4 Contiki Netstack VS mbed network stack 97

3.1 6LBR dual-interface subnets 100
3.2 STM32 Cube L1 software package block diagram 101
3.3 STM32 Cube L1 layers interaction 102
3.4 STM32 Nucleo L152-RE 105
3.5 STM32L152RE main connectors 106
3.6 STM32L152RE technical details 106
3.7 ST7580 powerline main protocols 109
3.8 STM32NucleoL152RE with SPIRIT1 110
3.9 STM32NucleoL152RE with SPIRIT1 and sensors from front 111
3.10 STM32NucleoL152RE with SPIRIT1 and sensors from side 111
3.11 STM32NucleoL152RE with ST7580 powerline modem . . . 112
3.12 STM32NucleoL152RE with ST7580 powerline and SPIRIT1 113
3.13 an example of a network with a virtual DODAG root . . . 120
3.14 RPL prefixes dissemination in a network with two subnets 123
3.15 Netstack vectorized 124
3.16 new layers interaction during a "receive-and-reply to a packet" event 125
3.17 Contiki uIP's main global buffers 130

4.1 3-interfaces device for memory sizes analysis 151
4.2 4-interfaces device for memory sizes analysis 151
4.3 original and dual-interface BR performances with SPIRIT1 162
4.4 original and dual-interface BR performances with ST7580 . 163
4.5 multicast DIO output performances comparison 165
4.6 BR RPL Instance Initialization Performances 168

List of Tables

1.1 radio frequencies allocation 17

1.2 LoRa details 18

2.1 main boards comparison 83

2.2 main IoT microcontrollers features 84

3.1 Linux kernel IPv6 routing table without Contiki 6LoWPAN 118

3.2 Linux kernel IPv6 routing table with Contiki 6LoWPAN . 118

4.1 RPL dual-interface border router memory sections sizes . . 151

4.2 n-interfaces device RAM sections sizes 152

4.3 original RPL border router memory sections sizes (bytes) . 152

4.4 border router FLASH sizes comparison (bytes) 152

4.5 border router RAM sizes comparison (bytes) 153

4.6 multi-interface BR RAM sizes comparison (bytes) 153

4.7 vectorized variables that greatly influence RAM overhead . 153

4.8 RPL UDP mono-interface client memory sections sizes . . 154

4.9 original RPL UDP client memory sections sizes 154

4.10 RPL UDP client FLASH sizes comparison (bytes) 154

4.11 RPL UDP client RAM sizes comparison (bytes) 155

4.12 RPL UDP mono-interface client with sensors memory sections sizes . . . 155

4.13 RPL UDP client with sensors without multi-interface changes memory sections sizes . . . 155

4.14 RPL UDP client with sensors FLASH sizes comparison (bytes) 156

4.15 RPL UDP client with sensors RAM sizes comparison (bytes) 156

4.16 RPL UDP mono-interface server memory sections sizes (bytes) 156

4.17 RPL UDP server FLASH sizes comparison (bytes) 157

4.18 RPL UDP server RAM sizes comparison (bytes) 157

4.19 Nucleo with SPIRIT1 Performances Comparison 161

4.20 Nucleo with ST7580 Performances Comparison 161

4.21 multi-interface feature performances comparison 164

4.22 multicast DIO output performances averages 165

4.23 BR RPL instance initialization performances averages . . . 167

A.1 GPIO conflicts 172

A.2 GPIO configuration 174

List of Source Code

2.1 the radio sleep cycle implemented with protothreads in pseudocode 86

2.2 MAC driver 89

3.1 Makefile changes 113

3.2 RPL DAG 120

3.3 prefix list vectorization 122

3.4 new Netstack arrays 126

3.5 dual-interface node Netstack arrays initialization 126

3.6 new Contiki source code organization 127

3.7 new multi-interface feature's main macros 127

3.8 new Netstack layers' functions invocation 128

3.9 interfaces identifiers declaration 129

3.10 interfaceID packetbuf attribute assignment to incoming packet 130

3.11 interfaceID packetbuf attribute usage 131

3.12 Interface structure vectorization 131

3.13 general functions adaptation 131

3.14 new unicast address structure with interface ID field 133

3.15 new multicast output implementation 134

3.16 new function: get next-hop interface ID 135

3.17 source address selection for a unicast destination in uip6.c 135

3.18 source address selection for a unicast destination in uip-icmp6.c 136
3.19 source address selection for a unicast destination in uip-nd6.c during NS input 137
3.20 source address selection for a unicast destination in uip-nd6.c during RS output 137
3.21 Source address selection for a unicast destination in uip-nd6.c during RA output 137
3.22 easy source address selection for multicast destination in uip-icmp6.c 138
3.23 easy source address selection for a unicast destination in uip-icmp6.c 139
3.24 RPL UDP server's addresses lists initialization 141
3.25 bash command line for obtaining an ELF file 145
3.26 bash command line for obtaining an ELF file for a platform with SPIRIT1 145
3.27 bash command line for obtaining an ELF file for a platform with sensors 146
3.28 bash command line for obtaining a binary from an ELF . . 146
3.29 copying the bin into the device folder 146
3.30 instantiate a RPL net via tunslip6 146
4.1 command line to obtain memory footprint 150
4.2 multi-interface BR RPl instance initialization with cycle counter 166
4.3 original BR RPl instance initialization with cycle counter . 167
A.1 dwt.h 175
A.2 dwt.c 175

Introduction

Thanks to the recent technological and scientific evolution, infinite new different possibilities are available in the Internet of Things world. Today, we are running towards a fully-connected reality in which devices will always more help humans, companies and state agencies in several daily affairs allowing them to save time, money and a lot of other different kind of resources. Different researches have been carried out to better understand the importance of IoT revolution: for example, [2] the Internet of Things Observatory of the School of Management of Politecnico di Milano estimates that a national pervasive adoption of smart lighting, smart mobility management and smart garbage collection applications could allow citizens, public administration and companies to save up to 4,2 billion of euros/ year, and that these applications could even improve cities liveability by avoiding the emission of 7,2 million tons of CO2/year, and also by avoiding, for every citizen, to be stuck in the traffic or looking for a free parking slot, saving the equivalent of 5 days every year. Other studies tried to estimate the wideness of IoT revolution. According to Gartner, Inc. (a technology research and advisory corporation), there will be nearly 26 billion devices on the Internet of Things by 2020 [1], while ABI Research estimates that more than 30 billion devices will be wirelessly connected to the Internet of Things by 2020 [3].

For all the above reasons, IoT is currently one of the main research trend, and great efforts will be put in this area also in the coming years. Different topics are currently still open and under deep investigations: one of the most important is surely related to the creation and the management of scalable, secure and heterogeneous networks in which devices can interact among them in a reliable and safe way. In order to implement a IoT system, low-power-oriented and memory-constrained-oriented Operating Systems must be utilized for the motes of a network. Many OS are available for this purpose, but the most interesting ones are the open-source OS. In fact, every IoT system needs customization in order to perform well, as these systems

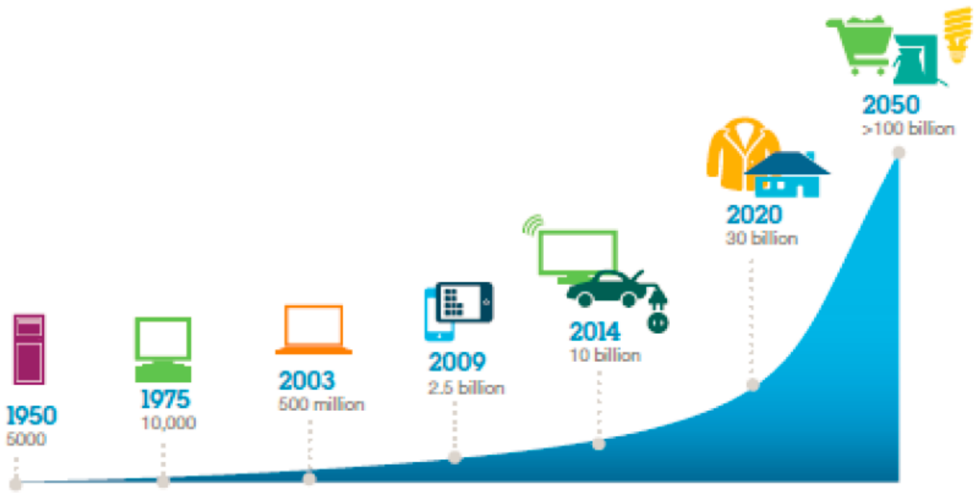


Figure 1: Internet of Things trend

are resource-constrained. Thus, the opportunity to modify the source-code in order to adapt the OS functionalities to a specific IoT system is very relevant. In order to develop a IoT system, it is advisable to follow as much as possible the standardization rules given by international organizations like IETF and IEEE. These organizations have defined protocols to be implemented in IoT systems's network stack in order to obtain efficiency and interoperability. In particular, interoperability is a hot topic in the IoT, as these IoT systems will be composed of hundreds, even thousands devices that will interoperate in order to furnish some services or accomplish some tasks. In this prospective, by considering the most diffused IoT-oriented OSs like Contiki, TinyOS and others, it has been noticed that they do not provide interoperability to devices with different physical interfaces. Thus, the main idea behind this work is to create a network prototype that permits interoperability between devices with different interfaces.

Thesis Goal

Goal of this thesis is to create a 6LoWPAN-based network prototype that exploits nodes with different communication interfaces. Such heterogeneous network configuration allows information exchange between several devices, which gives users remote management opportunities for several applications.

Thesis goal is to create a prototype to be adopted in home automation field and in industrial field. Possible home automation scenarios are composed by sensors that communicate with household appliances, with lightning control systems, with audio equipments (home theater, speakers..), with entertainment devices (televisions, monitors..) and so on. Possible industrial applications are composed of several sensors which communicate with actuators, with automatic doors, with robots and so on. Wireless Sensor Networks in general should benefit from 6LoWPAN multi-interface nodes, as it should add potential to already existing IoT application fields and even create new ones.

Furthermore, the prototype must be scalable, low-power and must have low footprint.

Outline

This thesis work is structured as follows:

Chapter One gives a formal definition of Internet of Things and introduces some underlying concepts that are needed to better focus the topic. The chapter goes on by illustrating some enabling technologies for the IoT and then some other relevant technologies that are needed to fully understand this work.

Chapter Two illustrates the hardware and software state of the art. First, Contiki (version 3.0), is introduced as the operating system selected to implement the network prototype developed. The network stack and the protocols already provided by Contiki are also discussed. Then, a brief overview of other real-time operating systems running on devices for LoWPAN networks is appended.

Chapter Three describes the developed solution, giving a brief overview of the hardware involved, explaining the obtained working configurations and detailing the introduced software modifications, which are exposed in detail and together with some source code listings.

Chapter Four illustrates some evaluation criteria of the proposed solution.

Chapter 1

Internet of Things

The Internet of Things is the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment [1].

As stated by the definition, these physical objects contain embedded technology of which some concrete examples will be given later; anyway, these objects communicate between them, usually via radio waves (RFID), to exchange data.

Starting from the above definition, there are some underlying technical visions for the IoT that need to be mentioned in order to comprehend what are the objectives in the IoT research field:

- the concept of "ubiquitous networks", which focuses on the communication aspects of technologies that are available anytime and everywhere
- the concept of "next-generation networks" (NGN), which are integrated core networks that are set to form the underlying platform for the services and applications of the future

Ubiquitous Networks

Literally, a ubiquitous networked environment is one in which networks are available everywhere and anytime [4]. Early forms of ubiquitous information and communication networks are evident in the widespread use of

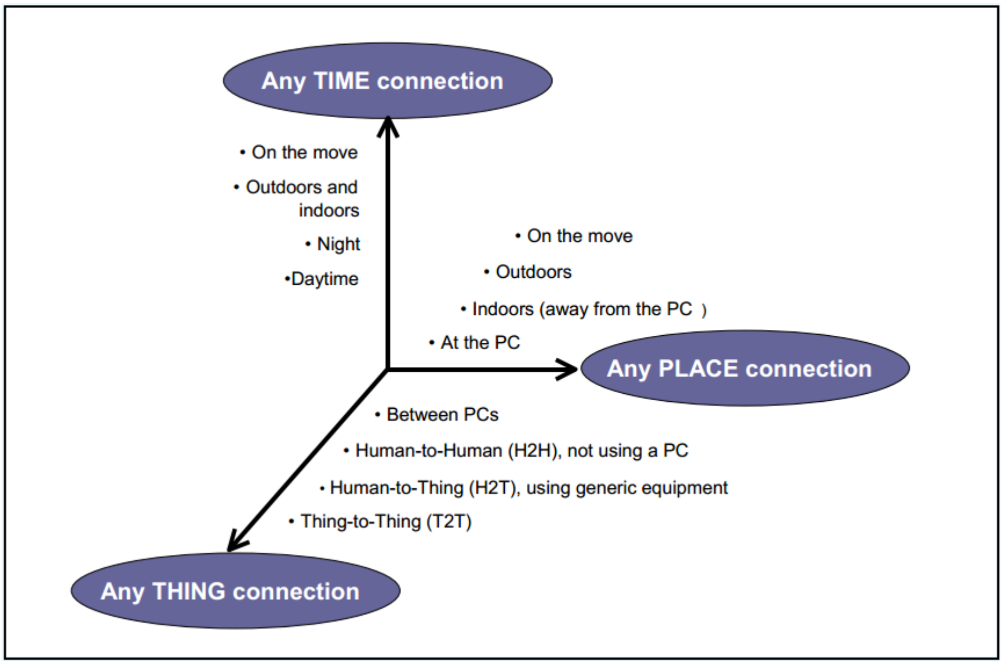


Figure 1.1: source: ITU, adapted from the Nomura Research Institute, "Ubiquitous Networking: Business Opportunities and Strategic Issues", August 2004

mobile phones: there were over 7,2 billion mobile phones in circulation by the end of 2014 [5]. Ubiquitous networks take mobile networks one step further, embedding short-range mobile transceivers into a wide array of additional gadgets and everyday items, enabling new forms of collaboration and communication between people and things, and between things themselves.

The next step in this technological revolution is to connect inanimate objects and things to communication networks. This is the vision of a truly ubiquitous network –"anytime, anywhere, by anyone and anything". Picture 1.1 gives some examples of "anytime, anywhere and anything". In particular, "anything" considers three main subjects: PCs, humans and things, where things are all the generic devices which differ for some aspect from PCs.

In this context, consumer products might be tracked using tiny radio transmitters or tagged with embedded hyperlinks and sensors.

Today, users can connect at any time and at any location; tomorrow's global networks will not only consist of humans and electronic devices, but all sorts of inanimate things as well; these things will be able to communicate with other things, e.g. fridges with grocery stores, laundry machines with clothing, implanted tags with medical equipment, and vehicles with stationary and moving objects. With the benefit of integrated information processing capacity, industrial products will take on smart characteristics and capabilities, e.g. electronic identities that can be queried remotely, or they could be equipped with sensors for detecting physical changes around them. Such developments will make the merely static objects of today into newly dynamic things, embedding intelligence in our environment, and stimulating the creation of innovative products and entirely new services.

Next-Generation Networks

Nowadays the underlying mobility of services remains limited: end-user services other than voice are hardly portable across networks, which is a central functionality in order to exploit thing-to-thing communications. Next-generation networks aim at offer a broader mobility. “Generalized mobility” is a term closely associated with NGN, which denotes the possibility of seamless and ubiquitous access to services, irrespective of location and the technology used. The fundamental difference between the networks of today and NGN will be the full transition they imply from current circuit-switched networks to packet-based systems such as those using IP. Future services enabled by NGN are expected to adapt to the needs of individual users (people and things), through real-time knowledge of their status and context: for instance, their availability and communication status (e.g. online, offline, busy). Multiple devices, telecommunication technologies, positioning and sensing systems, location-aware or context-aware applications, and so on, form the integral elements of a richer NGN communication environment. NGN will address both network and service elements, providing new opportunities for service providers, operators, content developers, manufacturers and users. The creation of the Internet of Things will entail the connection of everyday objects and devices to all kinds of networks, e.g. company intranet, peer-to-peer networks and even the global internet; for this reason, its development is expected to be of great impact on the telecommunication industry. It will challenge existing structures within established companies, and form the basis for entirely new opportunities and business models.

1.1 ITU Enabling Technologies

After having illustrate the underlying technical visions that show how the future will be in the next generation, the discussion moves on with the enabling technologies for the IoT choices by the ITU.

The International Telecommunication Union is a specialized agency of the United Nations (UN) that is responsible for issues that concern information and communication technologies. The creation of the Internet of Things depends on dynamic technical innovation in a number of important fields [4]:

- in order to connect everyday objects and devices to large databases and networks (to the internet), a simple, unobtrusive and cost-effective system of item identification is indispensable; only then can data about things be collected and processed. Radio-frequency identification (RFID) offers just such a possibility
- data collection can of course benefit from the ability to detect changes in the physical status of things, i.e. through sensor technologies
- embedded intelligence in the things themselves can further enhance the power of the network by devolving information processing capabilities to the edges of the network
- advances in miniaturization and nanotechnology mean that smaller and smaller things will have the ability to interact and connect

A combination of all of these developments will give rise to an Internet of Things that connects the world's objects in both a sensory and intelligent manner. Item-based tagging and identification will take anytime and anywhere communications to the next revolutionary step in networking: "anything communications". In the chapters that follow, the characteristics of each of the fields mentioned above will be illustrated in detail.

Radio-Frequency IDentifiers

RFID refers to those technologies that use radio waves to automatically identify and track specific items. Technically speaking, RFID systems consist of three main components [4]:

- A transponder or tag to carry data, which is located on the object to be identified; this normally consists of a coupling element (such as a coil, or microwave antenna) and an electronic microchip, less than 1/3 millimeter in size. Tags can be passive, semi-passive or active, based on their power source and the way they are used, and can be read-only, read/write or read/write/re-write, depending on how their data is encoded. Tags do not need an in-built power source, as they take the energy they need from the electro-magnetic field emitted by readers

- A reader or interrogator, which reads the transmitted data (e.g. on a device that is handheld or embedded in a wall)

- A middleware which forwards the data to another system, such as a database, a personal computer or robot control system for post-processing or for any other kind of use

In the most common type of system, the reader transmits a low-power radio signal to power on the tag (which, like the reader, has its own antenna). The tag then selectively reflects energy and thus transmits some data back to the reader, communicating its identity, location or any other relevant information. Most of the tags are passive, and are activated only when they are within the coverage area of the reader while, outside this area, they remain inactive.

One of the most pivotal aspects of these kind of electronic labels is that they enable the accurate identification of objects and the forwarding of this information, for example, to a database stored in the Cloud or on a remote server. In this manner, data and information processing capabilities can be potentially associated with any kind of object. This represents one of the first way to make not only people, but also things, connected and contactable.

Wireless Sensor Networks (WSNs)

A sensor is an electronic device which detects, senses or measures physical stimuli – like motion, heat or pressure – and converts them into an analogue or digital representation, allowing the raw data detected to be readable by machines or humans.

Sensors play a fundamental role in the building of a ubiquitous Internet of Things: they act as a bridge between the physical and virtual world. Some of then most common sensors' application areas include:

- Military – enemy tracking, battlefield surveillance etc.
- Environment – monitoring of humans or animals' habitat, behavior of birds, observation of environmental pollution and forecasting of natural disasters
- Healthcare – monitoring and tracking of patients and doctors and remote monitoring of physiological parameters, such as heart rate, level of substances in blood, care of elderly people
- Construction – monitoring of structural integrity
- Commercial applications – remote monitoring of the temperature of products
- Home applications – lighting, audio and video systems to security and kitchen appliances

When a sensor is part of a sensor network, it is known as a sensor "node". In simple terms, sensor nodes can be connected to each other in two ways:

- Wireline: these kind of connections provides high levels of security and reliability, and are appropriate whenever time-critical and mission-critical data and closed-loop control are required. However, laying cables and relocating them at a later date can be costly and time-consuming
- Wireless: Wireless sensor networks are generally less expensive, less visible and more flexible. A sensor node in a wireless sensor network is a small, low-power device, which normally includes sensors, a power-supply source, (optionally) a data storage unit, processing units (e.g. microcontrollers), a low-power radio module, analogue-to-digital converters (ADCs), data transceivers and controllers that tie all components together

One of the most important developments in sensor networks is the possibility for nodes to self-organize themselves into a network: in this way, information gathered and processed by a particular node identifies the nearest available node. This node receives the information and relays it on to a free peer, until the information reaches its ultimate destination.

Typically, sensor nodes are constrained by technical requirements like power consumption, size, memory and storage capacity. If the main function of an RFID tag is to identify an object and to track the location of a labeled product rapidly and accurately, i.e. to answer questions “what, which and where?”, then sensor technology provides information about the external environment and circumstances surrounding an object, thereby answering the question: “how?”.

The main distinguishing feature of an RFID sensor tag from a normal RFID tag is that, apart from tracking and monitoring functions, sensor-enabled RFID can function on the basis of data collected by the sensor. As such, it communicates information about momentous events and changes in physical conditions, and can take action (e.g. activate an alert).

The main technical challenges for RFID sensors are their limited processing speed, storage capacity and communication bandwidth; the effective processing and filtering of relevant information also needs to be addressed, as it might be costly to transmit the high volumes of data collected by sensors. In order to overcome these technical challenges, new hardware and software solutions are required, as is cost reduction.

Smart technologies

Current definitions of “smart” is very broad. The current trend is to define “smart” any conventional object that can react to external stimuli. Not only “classic “devices, such as PDAs and mobile phones, but also more “unusual“ things like, for instance, clothes. In this regard, some of the most interesting developments in this area include:

- Smart materials: incorporate sensors and actuators, as they sense stimuli and respond accordingly. Currently, there are three main kinds of smart materials:
  - Passive: respond directly and uniformly to stimuli without processing any of the signal
  - Active: with a remote controller can sense a signal and determine how to respond

  - Autonomous: carry fully integrated controllers, sensors and actuators

- Smart clothing and wearable computing: there is no clear boundary between smart clothing and wearable computing, although the following distinction may be drawn: in smart clothing, fabric remains the basic element; optical fibers or fibers than can conduct electricity can be woven between regular threads of fabric. By contrast, in the case of wearable computing, computing elements are the basis of the transformation: by miniaturizing size, decreasing weight and adding features such as durability or laundry-compliance, computers can be transformed into wearable computers. Wearable computing can be seen as "the result of a design philosophy that integrates embedded computation and sensing into everyday life to give users continuous access to the capabilities of personal computing"[6]. Wearable computers could also make use of smart fabrics, incorporating Global Positioning System (GPS), radio frequency and pressure detectors, temperature and shock sensors. Wearable computer hardware typically meets the following criteria:

  - The hardware should contain a microprocessor
  - The device should operate using software
  - The device is usually worn or supported on the body to enable hands-free computing
  - Ideally, the computer should always be accessible and ready to interact with its wearer through a wireline and/or wireless communication network

  Based on information about the external environment, the wearable computer is able to respond or take certain decisions such as adjusting to changes in temperature. Smart clothing and wearable computers can create intimate, responsive, interactive environments, enabling close human-to-machine interaction. Such closeness goes far beyond the simple proximity of the body to a device. The body itself could be used as a part of networking devices. This idea is at the heart of the concepts of the Personal Area Network (PAN), the Body Area Network (BAN) or the Human Area Network (HAN).

- Smart homes: networking and computing intelligence are penetrating into every corner of the home, from lighting, audio and video systems to security and kitchen appliances. The smart home is not just a collection of smart things, sensors and actuators, but an interconnected network of things, enabling voice or data-activated control from anywhere – through voicemail systems, internet, GPRS, SMS, mobile or fixed-line telephones from outside the home.

- Smart vehicles: automobiles today represent not only safe and comfortable means for traveling from one place to another, but also digital platforms for entertainment and access to information far beyond the traveling experience. The key technologies behind the smart vehicle have become known as "telematics". Automotive telematics is the blending of computers and telecommunications to enhance motor vehicles and provide convenient online services to road users through always-on connectivity. Using a myriad of smart materials, sensors, and other information technology solutions, the smart vehicle could avoid accidents, assess its own status, determine whether action needs to be taken, and if so, take it. It may even know how to escape to a safe haven in case of emergency. The intelligence involved in this sort of decision-making requires self-adaptability, self-sensing and memory.

- Robotics: automation is one of the key elements in the creation of a smarter world. Robots will be an integral part of such an environment. In general, a robot can be defined as an automated machine or a mechanical device that replaces human effort and which may, in some cases, mimic human or animal behavior or appearance. Despite our common perception of a robot as a mechanical creature that resembles a human being and imitates his/her behavior (a humanoid version), robots come in a wide variety of shapes and sizes. Robots equipped with sensors enabling them to sense and respond to stimuli will increasingly integrate into the networked world. Robotics introduces a greater degree of automation into everyday life, and will play a key role in the dawn of machine-to-machine interaction, in which data collected from the environment are forwarded to central processing points, in order for decisions to be taken with a minimum of human intervention. Development in personal robotics will also change the nature of human-to-machine interaction: robots will not only be assistants, but also friends and companions.

Nanotechnologies

Nanotechnologies are set to change the ICT industry dramatically, particularly reducing the size of data processing and storage devices. For the development of a truly ubiquitous and interactive Internet of Things, the combination of nanotechnologies and sensor technologies will be fundamental. Technical areas ripe for nanotechnologies include:

- Materials with properties like built-in chemical sensing or optical switching
- Medical developments like improved drug and gene therapies, biocompatible materials for implants or sensors for disease detection
- Environmental benefits like purification, artificial photosynthesis of clean energy and pollution control systems
- Information technologies, like quantum computing and computer chips that store trillions of bits of information on a device as small as the head of a pin

Besides the ITU enabling technologies, there are other technologies which need to be mentioned in order to fully comprehend the evolution in the IoT and some of the technical stuff of the project of this thesis.

1.2 Near-Field Communication (NFC)

Near field communication (NFC) is a set of communication protocols that enable two electronic devices, one of which is usually a portable device such as a smartphone, to establish communication by bringing them within a 4 cm range [6].

Evolved from radio frequency identification (RFID) tech, an NFC chip operates as one part of a wireless link. Once it is activated by another chip, small amounts of data between the two devices can be transferred when held a few centimeters from each other. NFC-enabled portable devices can be provided with applications, for example to read electronic tags or make payments when connected to an NFC-compliant apparatus. No pairing code is necessary to link up and because it uses chips that run on very low-power (or passive) devices, it is much more power-efficient than other wireless communication types. Each full NFC devices can work in three modes:

- NFC card emulation: NFC-enabled devices such as smartphones to act like smart cards, allowing users to perform transactions such as payment or ticketing
- NFC reader/writer: NFC-enabled devices read information stored on inexpensive NFC tags embedded in labels or smart posters
- NFC peer-to-peer: two NFC-enabled devices communicate with each other to exchange information in an ad-hoc fashion

The standards were provided by the NFC Forum [nfc-forum.org], which was responsible for promoting the technology and setting standards and certifies device compliance. Secure communications are available by applying encryption algorithms as is done for Credit Card. NFC standards cover communications protocols and data exchange formats and are based on existing radio-frequency identification (RFID) standards. NFC chips stocked inside credit cards for contact-less payments is nothing new. But a more recent and admittedly more enticing use case for NFC is with your smartphone, which can digitize your entire wallet.

1.3 ISM band

The industrial, scientific and medical (ISM) radio bands are reserved internationally for the use of radio frequency (RF) energy for industrial, scientific and medical purposes other than telecommunications. Examples of applications in these bands include radio-frequency process heating, microwave ovens, and medical diathermy machines. The powerful emissions of these devices can create electromagnetic interference and disrupt radio communication using the same frequency, so these devices were limited to certain bands of frequencies.

Despite the intent of the original allocations, and because there are multiple allocations, in recent years the fastest-growing uses of these bands have been for short-range, low power communications systems: cordless phones, Bluetooth devices, near field communication (NFC) devices, and wireless computer networks all use frequencies allocated to low power communications. The allocation of radio frequencies is provided according to Article 5 of the ITU Radio Regulations (edition 2012) and is illustrated in picture 1.1 on the next page.

where:

- Region 1 comprises Europe, Africa, the Middle East west of the Persian Gulf including Iraq, the former Soviet Union and Mongolia
- Region 2 covers the Americas, Greenland and some of the eastern Pacific Islands
- Region 3 contains most of non-former-Soviet-Union Asia, east of and including Iran, and most of Oceania

Type A frequency bands are designated for ISM applications. The use of these frequency bands for ISM applications shall be subject to special authorization by the administration concerned, in agreement with other administrations whose radio-communication services might be affected. In applying this provision, administrations shall have due regard to the latest relevant ITU-R Recommendations.

Type B frequency bands are also designated for ISM applications. Radio-communication services operating within these bands must accept harmful interference which may be caused by these applications.

In Europe, the use of the ISM band is covered by Short Range Device regulations issued by European Commission, based on technical recommendations by CEPT and standards by ETSI.

Wireless LAN devices use wavebands as follows:

- Bluetooth 2450 MHz band falls under WPAN
- HIPERLAN 5800 MHz band
- IEEE 802.11/Wi-Fi 2450 MHz and 5800 MHz bands
- IEEE 802.15.4, ZigBee and other personal area networks, which will be illustrated later, may use the 868 MHz, 915 MHz and 2450 MHz ISM bands because of frequency sharing between different allocations

Table 1.1: radio frequencies allocation

Center frequency	Type	Availability	Licensed users
6.78 MHz	A	Subject to local acceptance	fixed service and mobile service
13.56 MHz	B	Worldwide	fixed and Mobile services except Aeronautical mobile (R) service
27.12 MHz	B	Worldwide	fixed and mobile service except Aeronautical mobile service
40.68 MHz	B	Worldwide	fixed, Mobile services and Earth exploration-satellite service
433.92 MHz	A	only in Region 1, subject to local acceptance	amateur servide and radiolocation service, additional apply the provisions of footnote 5.280
915 MHz	B	Region 2 only (with some exceptions)	fixed, Mobile except aeronautical mobile and Radiolocation service; in Region 2 additional Amateur service
2.45 GHz	B	Worldwide	fixed, mobile, radiolocation, amateur and Amateur-satellite service
5.8 GHz	B	Worldwide	fixed-satellite, radiolocation, radiolocation, Amateur and Amateur-satellite service
24.125 GHz	B	Worldwide	amateur, amateur-satellite, radiolocation and earth exploration-satellite service (active)
61.25 GHz	A	Subject to local acceptance	fixed, intersatellite, mobile and radiolocation service
122.5 GHz	A	Subject to local acceptance	earth exploration-satellite (passive), fixed, intersatellite, mobile, space researhc (passive) and Amateur service
245 GHz	A	Subject to local acceptance	radiolocation, radio astronomy, amateur and amateur-satellite service

Table 1.2: LoRa details

Specification/feature	LoRa Support
Range	2-5 Km in dense urban and 15 Km in suburban areas
Frequency band	ISM band 868 MHz and 915 MHz
Standard	IEEE 802.15.4g
Modulation	spread spectrum modulation type is used which uses wide-band linear FM pulses. The frequency increase or frequency decrease over certain period is used to encode data information to be transmitted. It gives 30dB improvement over FSK.
Capacity	One LoRa gateway takes care of thousands of nodes.
Battery	Longer battery life
LoRa Physical layer	Takes care of frequency power, modulation, signaling between nodes and gateway.

1.4 LoRa

LoRa [7] stands for Long Range Radio. It is the wireless technology mainly targeted for Machine to Machine (aka M2M, i.e. direct communication between devices using any communications channel, including wired and wireless) and IoT networks. This technology will enable public or multi tenant networks to connect multiple applications running in the same network. This LoRa technology will fulfill to develop smart city with the help of LoRa sensors and automated products/applications. A LoRa network can be arranged to provide coverage similar to that of a cellular network [8]. Indeed many LoRa operators are cellular network operators who will be able to use existing masts to mount LoRa antennas. In some instances the LoRa antennas may be combined with cellular antennas as the frequencies may be close and combining antennas will provide significant cost advantages.

LoRa technology [9] is a proprietary wireless technology developed by Semtech Corporation. It utilizes a spread spectrum modulation in the Sub-GHz band to enable long range (greater than 10 miles) coverage, low power consumption (up to 10 years battery power), high network capacity (up to 1 million nodes), robust communication, and localization capability.

The LoRa protocol is a physical layer (PHY- OSI Layer 1) wireless component. Details of the protocol are listed in table 1.2 [7].

1.5 Wi-Fi

The Wi-Fi Alliance defines Wi-Fi as any "wireless local area network" (WLAN) product based on the IEEE 802.11 standards. However, the term "Wi-Fi" is used in general as a synonym for "WLAN" since most modern WLANs are based on these standards. Many devices can use Wi-Fi, e.g. personal computers, video-game consoles, smartphones, digital cameras, tablet computers and digital audio players; these can connect to a network resource such as the Internet via a wireless network access point, which has a range of about 20 meters indoors and a greater range outdoors.

802.11b and 802.11g use the 2.4 GHz ISM band because of this choice of frequency band, 802.11b and g equipment may occasionally suffer interference from microwave ovens, cordless telephones, and Bluetooth devices. A Wi-Fi signal occupies five channels in the 2.4 GHz band. Any two channel numbers that differ by five or more, such as 2 and 7, do not overlap.

802.11a uses the 5 GHz band which, for much of the world, offers at least 23 non-overlapping channels rather than the 2.4 GHz ISM frequency band, where adjacent channels overlap.

1.6 Bluetooth

Bluetooth is a wireless technology standard for exchanging data over short distances (using short-wavelength UHF radio waves in the 2.4 GHz ISM band) from fixed and mobile devices. It was invented by telecom vendor Ericsson in 1994 and now managed by the Bluetooth Special Interest Group (SIG). The IEEE standardized Bluetooth as IEEE 802.15.1, but no longer maintains the standard.

Bluetooth is a standard wire-replacement communications protocol primarily designed for low-power consumption, with a short range based on low-cost transceiver microchips in each device. Because the devices use a radio (broadcast) communications system, they do not have to be in visual line of sight of each other, however a quasi optical wireless path must be viable. Range is power-class-dependent, but effective ranges vary in practice: officially, class-3 radios have a range of up to 1 meter, class-2, most commonly found in mobile devices, 10 meters, and class-1, primarily for industrial use cases, 100 meters.

Bluetooth is a packet-based protocol with a master-slave structure: one master may communicate with up to eight slaves in a network called piconet, which consists of two or more devices occupying the same physical channel; some examples of piconets include a cell phone connected to a computer or a laptop and a Bluetooth-enabled digital camera.

The main difference between Bluetooth and Wi-Fi is that Wi-Fi is intended as a replacement for high speed cabling for general local area network access in work areas (this category of applications is sometimes called Wireless Local Area Networks -WLAN-), while Bluetooth was intended for portable equipment and its applications (this category of applications is outlined as the Wireless Personal Area Network -WPAN-).

Bluetooth exists in many products, such as telephones, tablets, media players, robotics systems, handheld, laptops and console gaming equipment, and some high definition headsets, modems and watches.

Bluetooth Low Energy (BLE)

Bluetooth Low Energy (LE) (also called Bluetooth Smart or Version 4.0+ of the Bluetooth specification) is the power- and application-friendly version of Bluetooth that was built for the Internet of Things (IoT). The power-efficiency of Bluetooth with low energy functionality makes it perfect for devices that run for long periods on power sources such as coin cell batteries or energy-harvesting devices.

The Bluetooth SIG identifies a number of markets for low energy technology, particularly in the smart home, health, sport and fitness sectors. Cited advantages include:

- low power requirements, operating for "months or years" on a button cell
- small size and low cost
- compatibility with a large installed base of mobile phones, tablets and computers

In the next chapters are presented a few models and protocols that will be consistently utilized both practically and as references during this thesis work. In particular, the OSI model and the Internet protocol suite will be

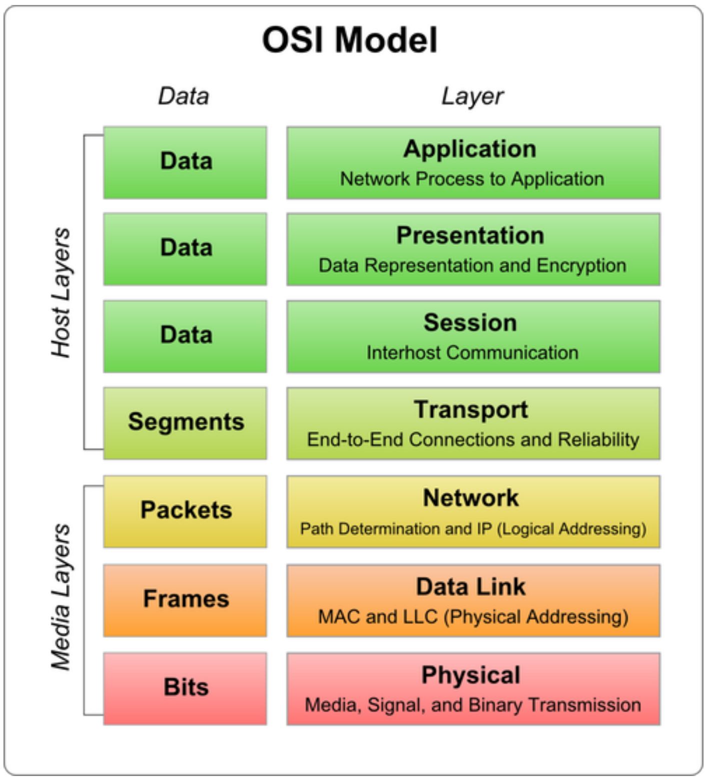

OSI Model
Data
Layer
Host Layers
Media Layers
Data
Application
Network Process to Application
Data
Presentation
Data Representation and Encryption
Data
Session
Interhost Communication
Segments
Transport
End-to-End Connections and Reliability
Packets
Network
Path Determination and IP (Logical Addressing)
Frames
Data Link
MAC and LLC (Physical Addressing)
Bits
Physical
Media, Signal, and Binary Transmission

Figure 1.2: OSI model

useful to understand the protocols utilized by Contiki and furthermore to fully comprehend the source code modifications to enable multiple interface communication between nodes.

1.7 OSI network stack

The Open Systems Interconnection model (OSI model) is a conceptual model that characterizes and standardizes the communication functions of a telecommunication or computing system without regard to their underlying internal structure and technology. Its goal is the interoperability of diverse communication systems with standard protocols. The model partitions a communication system into abstraction layers. The original version of the model defined seven layers. A layer serves the layer above it and is served by the layer below it.

Physical Layer

The physical layer has the following major functions:

- It defines the electrical and physical specifications of the data connection
- It defines the relationship between a device and a physical transmission medium (e.g., a copper or fiber optical cable, radio frequency). This includes the layout of pins, voltages, line impedance, cable specifications, signal timing and similar characteristics for connected devices and frequency (5 GHz or 2.4 GHz etc.) for wireless devices
- It defines transmission mode i.e. simplex, half duplex, full duplex
- It defines the network topology as bus, mesh, or ring being some of the most common
- Encoding of bits is done in this layer
- It determines whether the encoded bits will be transmitted by baseband (digital) or broadband (analog) signaling
- It mostly deals with raw data

The physical layer of Parallel SCSI operates in this layer, as do the physical layers of Ethernet and other local-area networks, such as Token Ring, FDDI, ITU-T G.hn, and IEEE 802.11 (Wi-Fi), as well as personal area networks such as Bluetooth and IEEE 802.15.4.

Data link layer

The goal of the data link layer is to provide reliable, efficient communication between adjacent machines connected by a single communication channel. Specifically:

- Group the physical layer bit stream into units called frames. Note that frames are nothing more than packets or messages. By convention, we'll use the term frames when discussing DLL packets

- Sender checksums the frame and sends checksum together with data. The checksum allows the receiver to determine when a frame has been damaged in transit
- Receiver recomputes the checksum and compares it with the received value. If they differ, an error has occurred and the frame is discarded
- Perhaps return a positive or negative acknowledgment to the sender. A positive acknowledgment indicate the frame was received without errors, while a negative acknowledgment indicates the opposite
- Flow control. Prevent a fast sender from overwhelming a slower receiver. For example, a supercomputer can easily generate data faster than a PC can consume it
- In general, provide service to the network layer. The network layer wants to be able to send packets to its neighbors without worrying about the details of getting it there in one piece

IEEE 802 divides the data link layer into two sublayers:[10]

- Media Access Control (MAC) layer - responsible for controlling how devices in a network gain access to medium and permission to transmit it
- Logical Link Control (LLC) layer - responsible for identifying Network layer protocols and then encapsulating them and controls error checking and frame synchronization.

The MAC and LLC layers of IEEE 802 networks such as 802.3 Ethernet, 802.11 Wi-Fi, and 802.15.4 ZigBee, operate at the data link layer.

Network layer

The network layer provides the functional and procedural means of transferring variable length data sequences (called datagrams) from one node to another connected to the same network. It translates logical network address into physical machine address. A network is a medium to which many nodes can be connected, on which every node has an address and which permits nodes connected to it to transfer messages to other nodes connected to it by merely providing the content of a message and the address of the destination node and letting the network find the way to deliver the message to the destination node, possibly routing it through intermediate nodes. If the message is too large to be transmitted from one node to another on the data link layer between those nodes, the network may implement message delivery by splitting the message into several fragments at one node, sending the fragments independently, and reassembling the fragments at another node. It may, but need not, report delivery errors.

Message delivery at the network layer is not necessarily guaranteed to be reliable. A network layer protocol may provide reliable message delivery, but it need not do so.

Transport layer

The transport layer provides the functional and procedural means of transferring variable-length data sequences from a source to a destination host via one or more networks, while maintaining the quality of service functions.

The transport layer controls the reliability of a given link through flow control, segmentation/desegmentation, and error control. Some protocols are state- and connection-oriented. This means that the transport layer can keep track of the segments and retransmit those that fail. The transport layer also provides the acknowledgement of the successful data transmission and sends the next data if no errors occurred. The transport layer creates packets out of the message received from the application layer. Packetizing is a process of dividing the long message into smaller messages. There are two primary transport layer protocols at present:

- Transmission Control Protocol (TCP): provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating over an IP network. TCP is the protocol that major Internet applications such as the World Wide Web, email, remote administration and file transfer rely on
- User Datagram Protocol (UDP): connectionless ("datagram") transport service. It has no handshaking dialogues and thus exposes the user's program to any unreliability of the underlying network protocol: there is no guarantee of delivery, ordering, or duplicate protection. The protocol provides checksums for data integrity and

port numbers for addressing different functions at the source and destination of the datagram

Session layer

The session layer controls the dialogues (connections) between computers. It establishes, manages and terminates the connections between the local and remote application. It provides for full-duplex, half-duplex, or simplex operation, and establishes checkpointing, adjournment, termination, and restart procedures. The OSI model made this layer responsible for graceful close of sessions, which is a property of the Transmission Control Protocol, and also for session checkpointing and recovery, which is not usually used in the Internet Protocol Suite. The session layer is commonly implemented explicitly in application environments that use remote procedure calls.

Presentation layer

The presentation layer establishes context between application-layer entities, in which the application-layer entities may use different syntax and semantics if the presentation service provides a big mapping between them. If a mapping is available, presentation service data units are encapsulated into session protocol data units, and passed down the protocol stack.

This layer provides independence from data representation (e.g., encryption) by translating between application and network formats. The presentation layer transforms data into the form that the application accepts. This layer formats and encrypts data to be sent across a network. It is sometimes called the syntax layer.

Application layer

The application layer is the OSI layer closest to the end user, which means both the OSI application layer and the user interact directly with the software application. This layer interacts with software applications that implement a communicating component. Such application programs fall outside the scope of the OSI model. Application-layer functions typically include identifying communication partners, determining resource availability, and synchronizing communication. When identifying communication partners, the application layer determines the identity and availability of communication partners for an application with data to transmit. When determining resource availability, the application layer must decide whether sufficient network or the requested communication exists. In synchronizing communication, all communication between applications requires cooperation that is managed by the application layer. This layer supports application and end-user processes. Communication partners are identified, quality of service is identified, user authentication and privacy are considered, and any constraints on data syntax are identified. Everything at this layer is application-specific.

1.8 Internet Protocol Suite

The Network Stack, also known as the Internet protocol suite is the computer networking model and set of communications protocols used on the Internet and similar computer networks. It is commonly known as TCP/IP, from its most important protocols, the Transmission Control Protocol (TCP) and the Internet Protocol (IP) were the first networking protocols defined in this standard.

The network stack provides end-to-end connectivity specifying how data should be packetized, addressed, transmitted, routed and received at the destination; these functionalities are organized in four abstraction layers which are used to sort all related protocols according to the scope of networking involved; from lowest to highest, the layers are:

- link layer: contains communication methods for data that remains within a single network segment (link)
- internet layer: connects independent networks, thus establishing internetworking
- transport layer: handles host-to-host communication
- application layer: provides process-to-process data exchange for applications

The TCP/IP model and many of its protocols are maintained by the Internet Engineering Task Force (IETF).

If we compare the Internet protocol suite with the OSI model, which is the standard about communication functions of telecommunication or computing systems, we notice that:

- the three top layers in the OSI model, i.e. the application layer, the presentation layer and the session layer, are not distinguished separately in the TCP/IP model, which only has an application layer above the transport layer
- the link layer is a combination of the data link layer (layer 2) and the physical layer (layer 1) of OSI model

In general, direct or strict comparisons should be avoided because the layering in TCP/IP is not a principal design criterion and in general is considered to be "harmful", as stated by RFC 3439 document. The IETF has repeatedly stated that Internet protocol and architecture development is not intended to be OSI-compliant.

IETF's RFC 1122 document loosely defines a four-layer model, with the layers having names, not numbers, as explained in the next chapters.

Application Layer

This layer contains the communications protocols and interface methods used in process-to-process communications across an Internet Protocol (IP) computer network. The application layer only standardizes communication and depends upon the underlying transport layer protocols to establish host-to-host data transfer channels and manage the data exchange in a client-server or peer-to-peer networking model. Though the TCP/IP application layer does not describe specific rules or data formats that applications must consider when communicating, the original specification (in RFC 1123) does rely on and recommend the robustness principle for application design.

There are two categories of application layer protocols: user protocols that provide service directly to users, and support protocols that provide common system functions. The most common Internet user protocols are Telnet (remote login), FTP (file transfer) and SMTP (electronic mail delivery).

Support protocols, used for host name mapping, booting, and management, include SNMP, BOOTP, RARP, and the Domain Name System (DNS) protocols.

Transport Layer

This layer performs host-to-host communications on either the same or different hosts and on either the local network or remote networks separated by routers; it provides a channel for the communication needs of applications and services such as connection-oriented data stream support, reliability, flow control, and multiplexing. The two primary transport layer protocols are TCP and UDP.

Internet Layer

This layer consists of a group of internetworking methods, protocols, and specifications that are used to transport datagrams (packets) from the originating host across network boundaries, if necessary, to the destination host specified by a network address (IP address) which is defined for this purpose by the Internet Protocol (IP). The internet layer derives its name from its function of forming an internet (uncapitalized), or facilitating internetworking, which is the concept of connecting multiple networks with each other through gateways. This layer defines the addressing and routing structures used for the TCP/IP protocol suite. The primary protocol in this scope is the Internet Protocol, which includes provision for addressing, type-of-service specification, fragmentation and reassembly, and security information. The datagram or connectionless nature of the IP protocol is a fundamental and characteristic feature of the Internet architecture. A common design aspect in the internet layer is the robustness principle: "Be liberal in what you accept, and conservative in what you send" as a misbehaving host can deny Internet service to many other users.

The internet layer has three basic functions:

- For outgoing packets, select the next-hop host (gateway) and transmit the packet to this host by passing it to the appropriate link layer implementation
- For incoming packets, capture packets and pass the packet payload up to the appropriate transport layer protocol, if appropriate

- Provide error detection and diagnostic capability

Link Layer

The link layer contains the group of methods and communications protocols that only operate on the link that a host is physically connected to. The link is the physical and logical network component used to interconnect hosts or nodes in the network and a link protocol is a suite of methods and standards that operate only between adjacent network nodes of a local area network segment or a wide area network connection. In particular, the link layer is a combination of the data link layer (layer 2) and the physical layer (layer 1) of the OSI model.

1.9 IEEE 802.15.4

IEEE 802.15.4 is a standard created and maintained by consultants which specifies the physical layer and media access control for low-rate wireless personal area networks (LR-WPANs). It is maintained by the IEEE 802.15 working group, which has defined it in 2003.

IEEE Std 802.15.4 defines the physical layer (PHY) and medium access control (MAC) sublayer specifications for low-data-rate wireless connectivity with fixed, portable, and moving devices with no battery or very limited battery consumption requirements typically operating in the personal operating space (POS) of 10 m. It is foreseen that, depending on the application, a longer range at a lower data rate may be an acceptable trade-off. [11] It can be contrasted with other approaches, such as Wi-Fi, which offer more bandwidth and require more power.

The standard defines two types of physical device [11, 12] :

- Full-Function Device (FFD): can talk to every node in the network (both FFD or Reduced-Function Devices) and can operate in three logical modes:
  - PAN coordinator: is fundamental to forming a new network, It may have an overall knowledge of the entire network, whatever that may be, but at all times it is responsible for network address allocation

  - Router: its primary role in the network is to route packets
  - End device: handles only communications and data transfer for itself

- Reduced-Function Devices (RFD): is intended for applications that are extremely simple, such as a light switch or an occupancy sensor; they generally communicate infrequently, spending most of their time in a quiescent state. An RFD may only associate with a single FFD at a time. Consequently, the RFD can be implemented using minimal resources and memory capacity.

Networks can be built as either peer-to-peer or star networks [11]; however, every network needs at least one FFD to work as the coordinator of the network. Networks are thus formed by groups of devices separated by suitable distances; each device has a unique 64-bit identifier, and if some conditions are met short 16-bit identifiers can be used within a restricted environment [12]; namely, within each PAN domain, communications will probably use short identifiers. 16 bit address space implies up to 65.535 nodes in a IEE 802.15.4 network.

The IEEE 802.15.4 specification supports many applications with MAC security requirements; however, if the networks are not secured, confidentiality, privacy, and integrity could be compromised.

The 802.15.4 security layer is handled at the media access control layer [11]; the application specifies its security requirements by setting the appropriate control parameters into the radio stack: if the application does not set any parameters, then security is not enabled by default.

The cryptographic mechanism in this standard is based on symmetric-key cryptography and uses keys that are provided by higher layer processes [11]; the establishment and maintenance of these keys are outside the scope of this standard. The cryptographic mechanism provides particular combinations of the following security services:

- Data confidentiality: Assurance that transmitted information is only disclosed to parties for which it is intended
- Data authenticity: Assurance of the source of transmitted information (and, hereby, that information was not modified in transit). This service is also known as Access control and message integrity
- Replay protection: Assurance that duplicate information is detected

Cryptographic frame protection may use a key shared between two peer devices (link key) or a key shared among a group of devices (group key), thus allowing so me flexibility and application-specific trade-offs between key storage and key maintenance costs versus the cryptographic protection provided. If a group key is used for peer-to-peer communication, protection is provided only against outsider devices and not against potential malicious devices in the key-sharing group.

The 802.15.4 specification defines eight different security suites [11]; we can classify them by the properties that they offer: no security, encryption only (AES-CTR), authentication only (AES-CBC-MAC) and encryption and authentication (AES-CCM). For each suite that offers encryption, the recipient can optionally enable replay protection.

In the IEEE 802 reference model of computer networking, the medium access control or media access control (MAC) layer is the lower sublayer of the data link layer (layer 2) of the seven-layer OSI model. The MAC sublayer provides addressing and channel access control mechanisms that make it possible for several terminals or network nodes to communicate within a multiple access network that incorporates a shared medium, e.g. an Ethernet network. The hardware that implements the MAC is referred to as a media access controller.

The MAC sublayer acts as an interface between the logical link control (LLC) sublayer and the network's physical layer. The MAC layer emulates a full-duplex logical communication channel in a multi-point network. This channel may provide unicast, multicast or broadcast communication service.

The MAC enables the transmission of MAC frames through the use of the physical channel; besides the data service, it offers a management interface and itself manages access to the physical channel and network beaconing; it also controls frame validation, guarantees time slots and handles node associations. Finally, it offers hook points for secure services.

The physical medium is accessed through a CSMA/CA protocol. Below are listed the principal IEEE 801.15.4 MAC and PHY layers features:

- Frequency bands:
  - 868.0–868.6 MHz: Europe, allows one communication channel
  - 902–928 MHz: North America, up to thirty channels
  - 2400–2483.5 MHz: worldwide use, up to sixteen channels.

- Data transfer rates: 20kbit/s, 40kbit/s, 100kbit/s and 250kbit/s
- Range: 10-20m
- Addressing: IEEE 64-bit addresses
- Network nodes: Up to 264 devices
- Security: 128 AES
- Channel access: CSMA-CA

CSMA/CA

Carrier sense multiple access (CSMA) is a probabilistic media access control (MAC) protocol in which a node verifies the absence of other traffic before transmitting on a shared transmission medium.

Carrier sense multiple access with collision avoidance (CSMA/CA) is a network multiple access method in which carrier sensing is used, but nodes attempt to avoid collisions by transmitting only when the channel is sensed to be "idle"; when they do transmit, nodes transmit their packet data in its entirety.

- Carrier Sense: prior to transmitting, a node first listens to the shared medium to determine whether another node is transmitting or not. First, it tries to detect the presence of a carrier signal from another node before attempting to transmit: if a carrier is sensed, the node waits for the transmission in progress to end before initiating its own transmission. In other words, CSMA is based on the principle "sense before transmit" or "listen before talk"
- Multiple access: multiple nodes may send and receive on the medium
- Collision avoidance: used to improve the performance of the CSMA method by attempting to divide the channel somewhat equally among all transmitting nodes within the collision domain. If another node was heard, it waits for a period of time for the node to stop transmitting before listening again for a free communications channel

The IEEE 802.15.4 LR-WPAN uses two types of channel access mechanism, depending on the network configuration [11]:

- Non-beacon-enabled PANs use an unslotted CSMA-CA channel access mechanism

- Beacon-enabled PANs use a slotted CSMA-CA channel access mechanism, where the backoff slots are aligned with the start of the beacon transmission; the backoff slots of all devices within one PAN are aligned to the PAN coordinator

1.10 CoAP

Constrained Application Protocol (CoAP) is an application layer protocol intended to be used in very simple electronics devices. It is particularly targeted for small low power sensors, switches, valves and similar components that need to be controlled or supervised remotely, through standard Internet networks; it is also intended for use in resource-constrained internet devices, such as WSN nodes.

CoAP adopts patterns from HTTP such as resource abstraction, URIs, RESTful interaction, and extensible header options, but uses a compact binary representations that are designed to be easy to parse. Unlike HTTP over TCP, CoAP uses UDP. This makes it possible to use CoAP in one-to-many and many-to-one communication patterns.

Central CoAP mechanisms are:

- Applications can send CoAP messages reliably (“confirmable”) or non-reliably (“non-confirmable”). Confirmables are retransmitted with exponential timeouts until acknowledged by the receiver or reaching the maximum number of retransmissions

- CoAP is intended to provide group communication via IP multicast

- CoAP features native push notifications through a publish/subscribe mechanism called "observing resources". Clients can send a request with an observe header option to a CoAP resource. The server keeps track of these subscribers and sends a response whenever the observed resource changes

1.11 ICMP

The Internet Control Message Protocol (ICMP) is a control protocol that is considered to be an integral part of IP, although it is architecturally layered upon IP, i.e., it uses IP to carry its data end-to-end just as a transport protocol like TCP or UDP does. ICMP provides error reporting, congestion reporting, and first-hop gateway redirection. ICMP differs from transport protocols such as TCP and UDP in that it is not typically used to exchange data between systems, nor it is regularly employed by end-user network applications (with the exception of some diagnostic tools like ping and traceroute). Some new ICMP control messages are exploited by IPv6 in order to setup the network. These new kind of messages are explained later.

1.12 IPv6

Definitions

LINK - RFC 4861 (Neighbor Discovery in IPv6): "a communication facility or medium over which nodes can communicate at the link layer, i.e., the layer immediately below IP. Examples are Ethernets (simple or bridged), PPP links, X.25, Frame Relay, or ATM networks as well as Internet-layer (or higher-layer) "tunnels", such as tunnels over IPv4 or IPv6 itself."

LINK - RFC 4903 (Multi-Link Subnet Issues): "generally used to refer to a topological area bounded by routers that decrement the IPv4 TTL or IPv6 Hop Limit when forwarding the packet."

ON-LINK ADDRESS: in RFC 5942 (IPv6 Subnet Model: The Relationship between Links and Subnet Prefixes): "In IPv6, an address is on-link (with respect to a specific LINK), if the address has been assigned to an interface attached to that LINK. Any node attached to the LINK can send a datagram directly to an on-link address without forwarding the datagram through a ROUTER."

in RFC 4861: "If a packet destination address is on-link, the next-hop address is that exactly address. Otherwise, the sender selects a router from the Default Router List."

INTERFACE - RFC 4861: "a node's attachment to a LINK."

ADDRESS - RFC 4861: "an IP-layer identifier for an INTERFACE or a set of interfaces"

ROUTER - RFC 4861: "a node that forwards IP packets not explicitly addressed to itself."

LINK-LAYER ADDRESS - RFC 4861: "a link-layer identifier for an INTERFACE. Examples include IEEE 802 addresses for Ethernet links."

PREFIX - RFC 4861: "a bit string that consists of some number of initial bits of an address."

SUBNET - RFC 4903: "generally used to refer to a topological area that uses the same address prefix, where that prefix is not further subdivided except into individual addresses."

NEIGHBORS - RFC 4861: "nodes attached to the same LINK."

LINK, ROUTER and ***SUBNET*** - RFC 4862 (IPv6 Stateless Address Autoconfiguration): "ROUTERS advertise prefixes that identify the SUBNET(S) associated with a LINK, while hosts generate an "interface identifier" that uniquely identifies an interface on a SUBNET. An address is formed by combining the two. In the absence of ROUTERS, a host can only generate link-local addresses. However, link-local addresses are sufficient for allowing communication among nodes attached to the same LINK."

Internet Protocol version 6 (IPv6) [13] is the most recent version of the Internet Protocol (IP), the communications protocol that provides an identification and location system for computers on networks and routes traffic across the Internet. IPv6 was developed by the Internet Engineering Task Force (IETF) to deal with the long-anticipated problem of IPv4 address exhaustion. IPv6 is intended to replace IPv4.

With the rapid growth of the Internet after commercialization in the 1990s, it became evident that far more addresses than the IPv4 address space were necessary to connect new devices in the future; by 1998, the IETF had formalized the successor protocol.

IPv6 uses a 128-bit address, theoretically allowing 2^{128}, or approximately $3,4 * 10^{38}$ addresses. The actual number is slightly smaller, as multiple ranges are reserved for special use or completely excluded from use.

The total number of possible IPv6 address is more than $7,9 * 10^{28}$ times as many as IPv4, which uses 32-bit addresses and provides approximately 4.3 billion addresses. The two protocols are not designed to be interoperable, complicating the transition to IPv6; however, several IPv6 transition mechanisms have been devised to permit communication between IPv4 and IPv6 hosts.

IPv6 provides other technical benefits in addition to a larger addressing space: in particular, it permits hierarchical address allocation methods that facilitate route aggregation across the Internet, and thus limit the expansion of routing tables.

Packet Header

In IPv6, the packet header and the process of packet forwarding have been simplified. Although IPv6 packet headers are at least twice the size of IPv4 packet headers, packet processing by routers is generally more efficient because less processing is required in routers. This furthers the end-to-end principle of Internet design, which envisioned that most processing in the network occurs in the leaf nodes.

The IPv6 header is not protected by a checksum. Integrity protection is assumed to be assured by both the link layer or error detection and correction methods in higher-layer protocols, such as TCP and UDP. The TTL field of IPv4 has been renamed to Hop Limit in IPv6, reflecting the fact that routers are no longer expected to compute the time a packet has

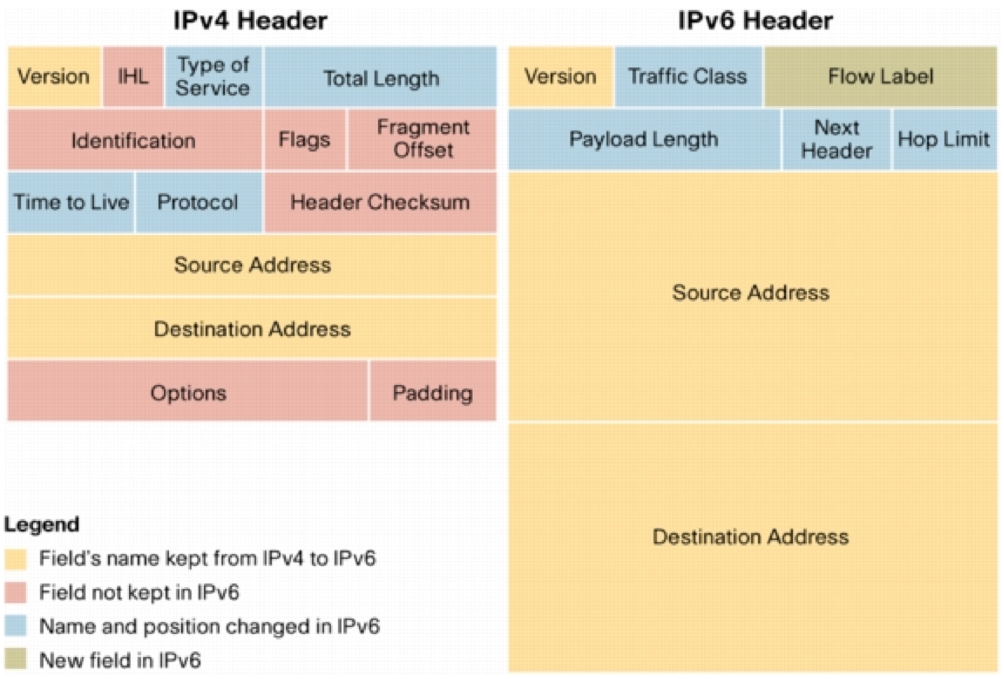


Figure 1.3: IPv6 header

spent in a queue.

An octet consists of 8 bits, i.e. a byte. Octets are often expressed and displayed using a variety of representations, for example in the hexadecimal, decimal, or octal number systems. The binary value of all 8 bits set (or turned on) is 11111111, equal to the hexadecimal value FF, the decimal value 255, and the octal value 377. One octet can be used to represent decimal values ranging from 0 to 255. Octets are used in the representation of Internet Protocol computer network addresses. An IPv4 address consists of four octets, usually shown individually as a series of decimal values ranging from 0 to 255, each separated by a full stop (dot). Using octets with all eight bits set, the representation of the highest numbered IPv4 address is 255.255.255.255. An IPv6 address consists of sixteen octets, shown using hexadecimal representation (two digits per octet) and using a colon character (:) after each pair of octet for readability for readability, like this FE80:0000:0000:0000:0123:4567:89AB:CDEF. The IPv6 packet header has a minimum size of 40 octets. Options are implemented as extensions. This provides the opportunity to extend the protocol in the future without affecting the core packet structure.

An IPv6 packet has two parts: a header and a payload. The header consists of a fixed portion with minimal functionality required for all packets and may be followed by optional extensions to implement special features. The fixed header occupies the first 40 octets (320 bits) of the IPv6 packet. It contains the source and destination addresses, traffic classification options, a hop counter, and the type of the optional extension or payload which follows the header. This Next Header field tells the receiver how to interpret the data which follows the header. If the packet contains options, this field contains the option type of the next option. The "Next Header" field of the last option, points to the upper-layer protocol that is carried in the packet's payload. Extension headers carry options that are used for special treatment of a packet in the network, e.g., for routing, fragmentation, and for security using the IPsec framework.

Unlike with IPv4, routers never fragment a packet. Hosts are expected to use Path MTU Discovery to make their packets small enough to reach the destination without needing to be fragmented.

IPv6 Addressing

In this paragraph is not discussed neither site-local addresses nor IPv4-Compatible IPv6 addresses as they are considered deprecated.

IPv6 addresses consist of two parts. The most-significant 64 bits are the subnet prefix to which the host is connected, and the least-significant 64 bits are the identifier of the host interface on the subnet. This means that the identifier need only be unique on the subnet to which the host is connected, which simplifies the detection of duplicate addresses.

There are three types of addresses:

- Unicast: An identifier for a single interface. A packet sent to a unicast address is delivered to the interface identified by that address.
- Anycast: An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to an anycast address is delivered to one of the interfaces identified by that address (the "nearest" one, according to the routing protocols' measure of distance).
- Multicast: An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to a multicast address is delivered to all interfaces identified by that address.

There are no broadcast addresses in IPv6, their function is superseded by multicast addresses.

IPv6 addresses of all types are assigned to interfaces, not nodes.

The preferred text representation form of addresses is x:x:x:x:x:x:x:x, where the 'x's are one to four hexadecimal digits of the eight 16-bit pieces of the address. Examples:

ABCD:EF01:2345:6789:ABCD:EF01:2345:6789

2001:DB8:0:0:8:800:200C:417A

Due to some methods of allocating certain styles of IPv6 addresses, it is common for addresses to contain long strings of zero bits. In order to make writing addresses containing zero bits easier, a special syntax is available to compress the zeros. The use of "::" indicates one or more groups of 16 bits of zeros. The "::" can only appear once in an address. The "::" can also be used to compress leading or trailing zeros in an address. For example, the following addresses:

a unicast address	2001:DB8:0:0:8:800:200C:417A
a multicast address	FF01:0:0:0:0:0:0:101
the loopback address	0:0:0:0:0:0:0:1
the unspecified address	0:0:0:0:0:0:0:0

may be represented as

a unicast address	2001:DB8:0:0:8:800:200C:417A
a multicast address	FF01::101
the loopback address	::1
the unspecified address	::

An IPv6 address prefix is represented by the notation:

ipv6-address/prefix-length

For example, the following are legal representations of the 60-bit prefix 20010DB80000CD3 (hexadecimal):

2001:0DB8:0000:CD30:0000:0000:0000:0000/60

2001:0DB8::CD30:0:0:0:0/60

2001:0DB8:0:CD30::/60

When writing both a node address and a prefix of that node address (e.g., the node's subnet prefix), the two can be combined as follows:

the node address	2001:0DB8:0:CD30:123:4567:89AB:CDEF
and its subnet number	2001:0DB8:0:CD30::/60
can be abbreviated as	2001:0DB8:0:CD30:123:4567:89AB:CDEF/60

The type of an IPv6 address is identified by the high-order bits of the address, as follows:

Address type	Binary prefix	IPv6 notation
Unspecified	00...0 (128 bits)	::/128
Loopback	00...1 (128 bits)	::1/128
Multicast	11111111	FF00::/8
Link-Local unicast	1111111010	FE80::/10
Global Unicast	(everything else)	

Unicast Addresses

An IPv6 unicast address refers to a single interface. Since each interface belongs to a single node, any of that node's interfaces' unicast addresses may be used as an identifier for the node. All interfaces are required to have at least one Link-Local unicast address. A single interface may also have multiple IPv6 addresses of any type (unicast, anycast, and multicast) or scope. There is one exception to this addressing model: A unicast address or a set of unicast addresses may be assigned to multiple physical interfaces if the implementation treats the multiple physical interfaces as one interface when presenting it to the internet layer. This is useful for load-sharing over multiple physical interfaces.

There are several types of unicast addresses in IPv6, in particular, Global Unicast and Link-Local unicast

IPv6 nodes may have considerable or little knowledge of the internal structure of the IPv6 address, depending on the role the node plays (for instance, host versus router). At a minimum, a node may consider that unicast addresses (including its own) have no internal structure:

128 bits
node address

A slightly sophisticated host (but still rather simple) may additionally be aware of subnet prefix(es) for the link(s) it is attached to, where different addresses may have different values for n:

n bits	128-n bits
subnet prefix	interface ID

Interface Identifiers

Interface identifiers in IPv6 unicast addresses are used to identify interfaces on a link. They are required to be unique within a subnet prefix. It is recommended that the same interface identifier not be assigned to different nodes on a link. They may also be unique over a broader scope. In some cases, an interface identifier is derived directly from that interface's link-layer address. The same interface identifier may be used on multiple interfaces on a single node, as long as they are attached to different subnets.

Note that the uniqueness of interface identifiers is independent of the uniqueness of IPv6 addresses. For example, a Global Unicast address may be created with a local scope interface identifier and a Link-Local address may be created with a universal scope interface identifier.

For all unicast addresses, except those that start with the binary value 000, Interface IDs are required to be 64 bits long and to be constructed in Modified EUI-64 format. Modified EUI-64 format-based interface identifiers may have universal scope when derived from a universal token (e.g., IEEE 802 48-bit MAC or IEEE EUI-64 identifiers) or may have local scope where a global token is not available or where global tokens are undesirable. Derivation method for Modified EUI-64 format is not deepen, as it does not affect this work.

Unspecified Address

The address 0:0:0:0:0:0:0:0 is called the unspecified address. It must never be assigned to any node. It indicates the absence of an address. One example of its use is in the Source Address field of any IPv6 packets sent by an initializing host before it has learned its own address.

Loopback Address

The unicast address 0:0:0:0:0:0:0:1 is called the loopback address. It may be used by a node to send an IPv6 packet to itself. It must not be assigned to any physical interface. It is treated as having Link-Local scope, and may be thought of as the Link-Local unicast address of a virtual interface (typically called the "loopback interface") to an imaginary link that goes nowhere.

Global Unicast Addresses

The general format for IPv6 Global Unicast addresses is as follows:

n bits	m bits	128-n-m bits
global routing prefix	subnet ID	interface ID

where the global routing prefix is a (typically hierarchically- structured) value assigned to a site (a cluster of subnets/links), the subnet ID is an identifier of a link within the site, and the interface ID is as defined as before illustrated.

Link-local Unicast Addresses

Link-Local addresses are for use on a single link. Link-Local addresses have the following format:

10 bits	54 bits	64 bits
1111111010	0	interface ID

Link-Local addresses are designed to be used for addressing on a single link for purposes such as automatic address configuration (illustrated in the next paragraphs), neighbor discovery (next paragraphs), or when no routers are present. Routers must not forward any packets with Link-Local source or destination addresses to other links.

IPv6 Addresses with Embedded IPv4 Addresses

The "IPv4-Mapped IPv6 address" was defined to assist in the IPv6 transition and is used to represent the addresses of IPv4 nodes as IPv6 addresses. The format is as follows:

80 bits	16 bits	32 bits
0000.......0000	FFFF	IPv4 address

Note: The IPv4 address used must be a globally-unique IPv4 unicast address.

Anycast Addresses

An IPv6 anycast address is an address that is assigned to more than one interface (typically belonging to different nodes), with the property that a packet sent to an anycast address is routed to the "nearest" interface having that address, according to the routing protocols' measure of distance. Anycast addresses are allocated from the unicast address space, using any of the defined unicast address formats. Thus, anycast addresses are syntactically indistinguishable from unicast addresses. When a unicast address is assigned to more than one interface, thus turning it into an anycast address, the nodes to which the address is assigned must be explicitly configured to know that it is an anycast address.

Multicast Addresses

An IPv6 multicast address is an identifier for a group of interfaces (typically on different nodes). An interface may belong to any number of multicast groups. Multicast addresses have the following format:

8 bits	4 bits	4 bits	112 bits
11111111	flgs	scop	group ID

flags is a set of 4 flags: | 0 | R | P | T |

The high-order flag is reserved, and must be initialized to 0.

T = 0 indicates a permanently-assigned ("well-known") multicast address, assigned by the Internet Assigned Numbers Authority (IANA).

T = 1 indicates a non-permanently-assigned ("transient" or "dynamically" assigned) multicast address.

scop is a 4-bit multicast scope value used to limit the scope of the multicast group. Some of the values are interface-Local, link-local, admin-local and global.

Interface-Local scope spans only a single interface on a node and is useful only for loopback transmission of multicast. Link-Local multicast scope spans the same topological region as the corresponding unicast scope. Admin-Local scope is the smallest scope that must be administratively configured, i.e., not automatically derived from physical connectivity or other, non-multicast-related configuration.

Group ID identifies the multicast group, either permanent or transient, within the given scope.

Pre-defined Multicast Addresses

The following well-known multicast addresses are pre-defined.

All Nodes Addresses:

FF01:0:0:0:0:0:0:1

FF02:0:0:0:0:0:0:1

The above multicast addresses identify the group of all IPv6 nodes, within scope 1 (interface-local) or 2 (link-local).

All Routers Addresses:

FF01:0:0:0:0:0:0:2

FF02:0:0:0:0:0:0:2

FF05:0:0:0:0:0:0:2

The above multicast addresses identify the group of all IPv6 routers, within scope 1 (interface-local) or 2 (link-local).

Solicited-Node Address:

FF02:0:0:0:0:1:FFXX:XXXX

Solicited-Node multicast address are computed as a function of a node's unicast and anycast addresses. A Solicited-Node multicast address is formed by taking the low-order 24 bits of an address (unicast or anycast) and appending those bits to the prefix FF02:0:0:0:0:1:FF00::/104 resulting in a multicast address in the range

FF02:0:0:0:0:1:FF00:0000

to

FF02:0:0:0:0:1:FFFF:FFFF

For example, the Solicited-Node multicast address corresponding to the IPv6 address 4037::01:800:200E:8C6C is FF02::1:FF0E:8C6C.

Solicited-node multicast address utilization is described in paragraph 1.14.

All RPL nodes addresses (link-local scope):

FF02::1a

Node's Required Addresses

A host is required to recognize the following addresses for identifying itself:

- Its required Link-Local address for each interface
- Any additional Unicast and Anycast addresses that have been configured for the node's interfaces (manually or automatically)
- The loopback address
- The All-Nodes multicast addresses
- The Solicited-Node multicast address for each of its unicast and anycast addresses
- Multicast addresses of all other groups to which the node belongs

A router is required to recognize all addresses that a host is required to recognize, plus the following addresses as identifying itself:

- The Subnet-Router Anycast addresses for all interfaces for which it is configured to act as a router
- All other Anycast addresses with which the router has been configured
- The All-Routers multicast addresses

Default Address Selection

The IPv6 addressing architecture allows multiple unicast addresses to be assigned to interfaces. These addresses may have different reachability scopes (link-local or global) and may also be "preferred" or "deprecated". Privacy considerations have introduced the concepts of "public addresses" and "temporary addresses". The mobility architecture introduces "home addresses" and "care-of addresses". The end result is that IPv6 implementations is very often faced with multiple possible source and destination addresses when initiating communication. It is desirable to have default algorithms, common across all implementations, for selecting source and destination addresses so that developers and administrators can reason about and predict the behavior of their systems.

Source Address Selection

The source address selection algorithm uses the concept of a "candidate set" of potential source addresses for a given destination address. The candidate set is the set of all addresses that could be used as a source address; the source address selection algorithm picks an address out of that set. It is recommended that the candidate source addresses be the set of unicast addresses assigned to the interface that will be used to send to the destination. (The "outgoing" interface.) On routers, the candidate set may include unicast addresses assigned to any interface that forwards packets.

[1]

For multicast and link-local destination addresses, the set of candidate source addresses must only include addresses assigned to interfaces belonging to the same link as the outgoing interface.

In any case, anycast addresses, multicast addresses, and the unspecified address must not be included in a candidate set.

The source address selection algorithm produces as output a single

[1]Contiki's source address selection algorithm (SAS in the following) refers to an obsoleted RFC document [14] that has been updated by the RFC 6724 [15]. In this paragraph, is presented the obsoleted version in order to understand Contiki's implementation. Anyway, the main concepts that are here reported are still valid in the updated version of sas.

source address for use with a given destination address. This algorithm only applies to IPv6 destination addresses, not IPv4 addresses. In this algorithm, the most relevant rules in order of relevance are the following:

- Prefer the same address: If SA = D, then prefer SA. Similarly, if SB = D, then prefer SB
- Prefer appropriate scope: If Scope(SA) < Scope(SB): If Scope(SA) < Scope(D), then prefer SB and otherwise prefer SA
- Avoid deprecated addresses
- Prefer outgoing interface: If SA is assigned to the interface that will be used to send to D and SB is assigned to a different interface, then prefer SA
- Use longest matching prefix: If CommonPrefixLen(SA, D) > CommonPrefixLen(SB, D), then prefer SA

The specification of source address selection assumes that routing (more precisely, selecting an outgoing interface on a node with multiple interfaces) is done before source address selection.

For example, suppose a node has interfaces on two different links, with both links having a working default router. Both of the interfaces have preferred global addresses. When sending to a global destination address, if there's no routing reason to prefer one interface over the other, then an implementation may preferentially choose the outgoing interface that will allow it to use the source address that shares a longer common prefix with the destination.

Implementations may also use the choice of router to influence the choice of source address. For example, suppose a host is on a link with two routers. One router is advertising a global prefix A and the other router is advertising global prefix B. Then when sending via the first router, the host may prefer source addresses with prefix A and when sending via the second router, prefer source addresses with prefix B.

Destination Address Selection

The destination address selection algorithm [45] takes a list of destination addresses and sorts the addresses to produce a new list. The most important rules of this algorithm are:

- Avoid unusable destinations: If DB is known to be unreachable or if Source(DB) is undefined, then prefer DA
- Prefer matching scope
- Avoid deprecated addresses
- Prefer smaller scope
- Use longest matching prefix

1.13 6LoWPAN

6LoWPAN [10, 16] is an acronym of IPv6 over Low power Wireless Personal Area Networks; 6LoWPAN is the name of the working group in the internet area of IETF and a specification to allow the use of IPv6 over IEEE 802.15.4 networks. Low-power wireless personal area networks (LoWPANs) comprise devices that conform to the IEEE 802.15.4-2003 standard by the IEEE. IEEE 802.15.4 devices are characterized by short range, low bit rate, low power, and low cost. Many of the devices employing IEEE 802.15.4 radios are limited in their computational power, memory, and/or energy availability. A LoWPAN is a simple low cost communication network that allows wireless connectivity in applications with limited power and relaxed throughput requirements. A LoWPAN typically includes devices that work together to connect the physical environment to real-world applications, e.g., wireless sensors.

Some of the characteristics of LoWPANs are as follows:

- Small packet size. Given that the maximum physical layer packet (MTU) with 802.15.4 is 127 bytes, the resulting maximum frame size at the media access control layer is 102 octets, as 802.15.4 maximum frame overhead is of 25 bytes. Link-layer security imposes further overhead up to 21 bytes, which in the maximum case leaves 81 octets for data packets. IP header then is 40 bytes, and 8 byte are required by UDP header. Thus, the remaining bytes for actual data are only 33

- Support for both 16-bit short or IEEE 64-bit extended media access control addresses
- Low bandwidth. Data rates of 250 kbps, 40 kbps, and 20 kbps for each of the currently defined physical layers (2.4 GHz, 915 MHz, and 868 MHz, respectively)
- Topologies include star and mesh operation
- Low power. Typically, some or all devices are battery operated
- Low cost. These devices are typically associated with sensors, switches, etc. This drives some of the other characteristics such as low processing, low memory, etc. Numerical values for "low" elided on purpose since costs tend to change over time
- Large number of devices expected to be deployed during the lifetime of the technology. This number is expected to dwarf the number of deployed personal computers, for example
- Location of the devices is typically not predefined, as they tend to be deployed in an ad-hoc fashion. Furthermore, sometimes the location of these devices may not be easily accessible. Additionally, these devices may move to new locations
- Devices within LoWPANs tend to be unreliable due to variety of reasons: uncertain radio connectivity, battery drain, device lockups, physical tampering, etc
- In many environments, devices connected to a LoWPAN may sleep for long periods of time in order to conserve energy, and are unable to communicate during these sleep periods

6LoWPAN operates packet fragmentation below the network layer: in fact, this protocol logically add a layer called “adaptation layer” between the network layer and the MAC layer of the TCP/IP stack. The header compression compress IP addresses when they can be derived from other headers, such as the 802.15.4 MAC header. For instance, it compress prefix for link-local addresses (fe80::) and elide address completely when it can be fully derived from the link-layer address. Furthermore, it compress common headers, as TCP, UDP and ICMP.

Devices in 6LoWPAN can be divided in two groups as mentioned in IEEE 802.15.4 section: full function and reduced function devices. FFDs

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version| Traffic Class |           Flow Label                  | IPv6
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  |
|         Payload Length        |  Next Header  |   Hop Limit   |  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  |
|                                                               |  |
+                                                               +  |
|                                                               |  |
+                         Source Address                        +  |
|                                                               |  |
+                                                               +  |
|                                                               |  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  |
|                                                               |  |
+                                                               +  |
|                                                               |  |
+                      Destination Address                      +  |
|                                                               |  |
+                                                               +  |
|                                                               |  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        | UDP
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  |
|            Length             |           Checksum            |  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1.4: IPv6 plus UDP header

typically have more resources and may be main powered. Accordingly, FFDs aid RFDs by providing functions such as network coordination, packet forwarding, interfacing with other types of networks, etc. In figure 1.4 is illustrated a IPv6 plus UDP header.

A standard IPv6/UDP header is 48 bytes in length. Figure 1.5 on the facing page gives an example of 6LoWPAN/UDP header in its simplest form (equivalent to the lower packet in Figure 1.6 on the next page), with a dispatch value and IPv6 header compression (LOWPAN IPHC), all IPv6 fields compressed, then followed by a UDP next-header compression byte (LOWPAN NHC) with compressed source and destination port fields and the UDP checksum (4 bytes).

Therefore in the likely best case the 6LoWPAN/UDP header is just 6 bytes in length.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Dispatch and LOWPAN_IPHC     |   LOWPAN_NHC  |  Src  |  Dst  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          UDP Checksum         | ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1.5: 6LoWPAN header in its simplest form

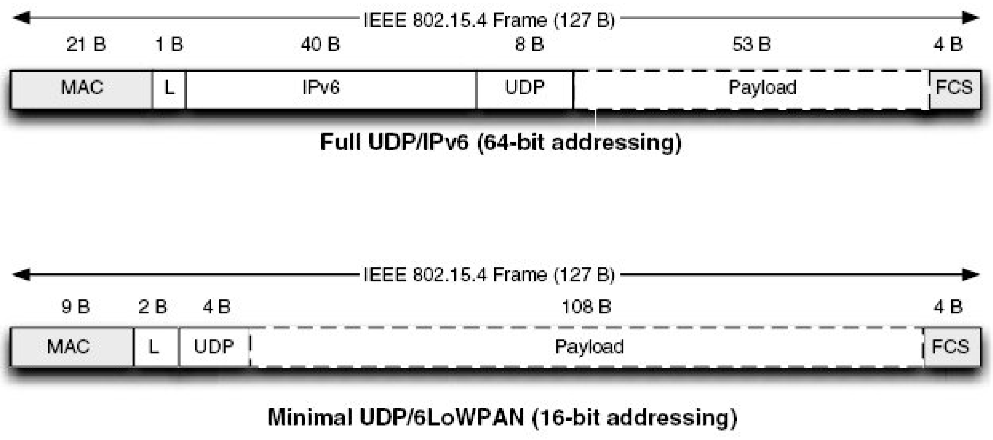

IEEE 802.15.4 Frame (127 B)
21 B
1 B
40 B
8 B
53 B
4 B
MAC
L
IPv6
UDP
Payload
FCS
Full UDP/IPv6 (64-bit addressing)
IEEE 802.15.4 Frame (127 B)
9 B
2 B
4 B
108 B
4 B
MAC
L
UDP
Payload
FCS
Minimal UDP/6LoWPAN (16-bit addressing)

Figure 1.6: IEEE 802.15.4 frame with 6LoWPAN/UDP header

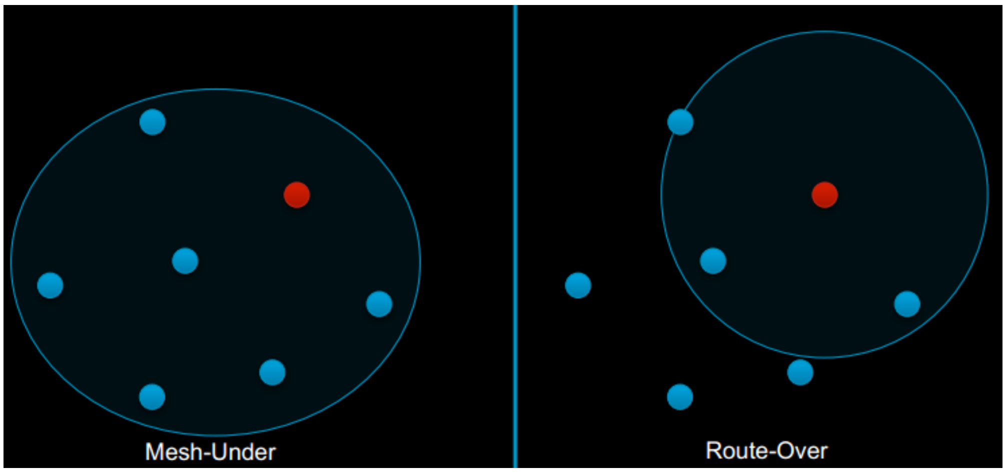


Figure 1.7: LLN: Mesh VS RouteOver

Mesh-Under and Route-Over Routing

In general, there are two categories of routing protocols [17, 18]: the mesh-under and the route-over. Figure 1.7 shows layers on which the routing decision occurs in TCP/IP protocol stack for 6LoWPAN. For mesh-under scheme the routing decision is taken in adaptation layer, whereas for route-over scheme the decision is taken in network layer.

In mesh-under scheme, the network layer does not perform any IP routing inside a LoWPAN. The adaptation layer performs the mesh routing and forwards packets to the destination over multiple radio hops. In mesh-under scheme, routing and forwarding are performed at link layer based on 802.15.4 frame or the 6LoWPAN header. To send a packet to a particular destination, the EUI 64 bit address or the 16 bit short address is used and sent it to a neighbor node to move the packet closer to the destination. Multiple link layer hops are used to complete a single IP hop and so it is called mesh-under. The networks which should adopts mesh-under protocol should present these characteristics:

- Single broadcast domain: All devices appear directly attached to the same broadcast medium and the IP layer can transmit a datagram to any number of devices attached to the same link. More formally, all communication is transitive within a single broadcast domain (if A can send to B and B can send to C, then A can send to C)
- Deterministic link characteristics: any differences in link characteris-

tics (e.g. communication latency, throughput, and loss rates) between different node pairs attached to the same link are insignificant. From a routing perspective, the link cost rarely varies

- High reliability: the link delivers messages to their intended destination with relatively high reliability

Some of the issues that arise with tentative to exploit mesh-under routing protocol in LLNs are reported here [36]. Mesh-under routing is exploited by ethernet networks.

In route-over scheme all routing decisions are taken in the network layer where each node acts as an IP router. In route-over, each link layer hop is an IP hop. The IP routing supports the forwarding of packets between these links. In the forwarding process IP routing tables and IPv6 hop-by-hop options are used. For routing and forwarding processes the network layer takes decision using the additional encapsulated IP header. This kind of protocol was adopted by RPL, the standard routing protocol for LLN.

1.14 Neighbor Discovery Protocol for IPv6

Nodes (hosts and routers) use Neighbor Discovery for IPv6 (ND6) [19] to determine the link-layer addresses for neighbors known to reside on attached links and to quickly purge cached values that become invalid. Hosts also use Neighbor Discovery to find neighboring routers that are willing to forward packets on their behalf. Finally, nodes use the protocol to actively keep track of which neighbors are reachable and which are not, and to detect changed link-layer addresses. When a router or the path to a router fails, a host actively searches for functioning alternates.

This protocol solves a set of problems related to the interaction between nodes attached to the same link. It defines mechanisms for solving each of the following problems:

- Router Discovery: How hosts locate routers that reside on an attached link

- Prefix Discovery: How hosts discover the set of address prefixes that define which destinations are on-link for an attached link (Nodes use prefixes to distinguish destinations that reside on-link from those only reachable through a router)

- Parameter Discovery: How a node learns link parameters (such as the link MTU) or Internet parameters (such as the hop limit value) to place in outgoing packets
- Address Autoconfiguration: Introduces the mechanisms needed in order to allow nodes to configure an address for an interface in a stateless manner. Stateless address autoconfiguration is explained in the next paragraphs
- Address resolution: How nodes determine the link-layer address of an on-link destination (e.g., a neighbor) given only the destination's IP address
- Next-hop determination: The algorithm for mapping an IP destination address into the IP address of the neighbor to which traffic for the destination should be sent. The next- hop can be a router or the destination itself
- Neighbor Unreachability Detection: How nodes determine that a neighbor is no longer reachable. For neighbors used as routers, alternate default routers can be tried. For both routers and hosts, address resolution can be performed again
- Duplicate Address Detection: How a node determines whether or not an address it wishes to use is already in use by another node
- Redirect: How a router informs a host of a better first-hop node to reach a particular destination

ICMPv6 Protocol Packets

Neighbor Discovery defines five different ICMP packet types: A pair of Router Solicitation and Router Advertisement messages, a pair of Neighbor Solicitation and Neighbor Advertisements messages, and a Redirect message. The messages serve the following purpose:

- Router Solicitation: When an interface becomes enabled, hosts may send out Router Solicitations that request routers to generate Router Advertisements immediately rather than at their next scheduled time

- Router Advertisement: Routers advertise their presence together with various link and Internet parameters either periodically, or in response to a Router Solicitation message. Router Advertisements contain prefixes that are used for determining whether another address shares the same link (on-link determination) and/or address configuration, a suggested hop limit value, etc. In a RA, in the option fields of the packet can be present the Prefix Information Option. These options specify the prefixes that are on-link and/or are used for stateless address autoconfiguration. A router should include all its on-link prefixes (except the link-local prefix) so that multihomed hosts have complete prefix information about on-link destinations for the links to which they attach. If complete information is lacking, a host with multiple interfaces may not be able to choose the correct outgoing interface when sending traffic to its neighbors

- Neighbor Solicitation: Sent by a node to determine the link-layer address of a neighbor, or to verify that a neighbor is still reachable via a cached link-layer address. Neighbor Solicitations are also used for Duplicate Address Detection

- Neighbor Advertisement: A response to a Neighbor Solicitation message. A node may also send unsolicited Neighbor Advertisements to announce a link-layer address change

- Redirect: Used by routers to inform hosts of a better first hop for a destination

On multicast-capable links, each router periodically multicasts a Router Advertisement packet announcing its availability. A host receives Router Advertisements from all routers, building a list of default routers.

Router Advertisements contain a list of prefixes used for on-link determination and/or autonomous address configuration; flags associated with the prefixes specify the intended uses of a particular prefix. Hosts use the advertised on-link prefixes to build and maintain a list that is used in deciding when a packet's destination is on-link or beyond a router. Note that a destination can be on-link even though it is not covered by any advertised on- link prefix. In such cases, a router can send a Redirect informing the sender that the destination is a neighbor.

Nodes accomplish address resolution by multicasting a Neighbor Solicitation that asks the target node to return its link-layer address. Neighbor

Solicitation messages are multicast to the solicited-node multicast address of the target address. The target returns its link-layer address in a unicast Neighbor Advertisement message. A single request-response pair of packets is sufficient for both the initiator and the target to resolve each other's link-layer addresses; the initiator includes its link-layer address in the Neighbor Solicitation.

Neighbor Unreachability Detection detects the failure of a neighbor or the failure of the forward path to the neighbor. Doing so requires positive confirmation that packets sent to a neighbor are actually reaching that neighbor and being processed properly by its IP layer. When positive confirmation is not forthcoming through such "hints", a node sends unicast Neighbor Solicitation messages that solicit Neighbor Advertisements as reachability confirmation from the next hop.

Data Structures

Hosts need to maintain the following pieces of information for each interface:

- Neighbor Cache: A set of entries about individual neighbors to which traffic has been sent recently. Entries are keyed on the neighbor's on-link unicast IP address and contain such information as its link-layer address, a flag indicating whether the neighbor is a router or a host, a pointer to any queued packets waiting for address resolution to complete, etc. A Neighbor Cache entry also contains information used by the Neighbor Unreachability Detection algorithm, including the reachability state, the number of unanswered probes, and the time the next Neighbor Unreachability Detection event is scheduled to take place
- Destination Cache: A set of entries about destinations to which traffic has been sent recently. The Destination Cache includes both on-link and off-link destinations and provides a level of indirection into the Neighbor Cache; the Destination Cache maps a destination IP address to the IP address of the next-hop neighbor. This cache is updated with information learned from Redirect messages. Implementations may find it convenient to store additional information not directly related to Neighbor Discovery in Destination Cache entries, such as the Path MTU (PMTU) and round-trip timers maintained by transport protocols

- Prefix List - A list of the prefixes that define a set of addresses that are on-link (see paragraph [Definitions] for on-link address definition). Prefix List entries are created from information received in Router Advertisements. Each entry has an associated invalidation timer value (extracted from the advertisement) used to expire prefixes when they become invalid. A special "infinity" timer value specifies that a prefix remains valid forever, unless a new (finite) value is received in a subsequent advertisement. The link-local prefix is considered to be on the prefix list with an infinite invalidation timer regardless of whether routers are advertising a prefix for it

- Default Router List: A list of routers to which packets may be sent. Router list entries point to entries in the Neighbor Cache; the algorithm for selecting a default router favors routers known to be reachable over those whose reachability is suspect. Each entry also has an associated invalidation timer value (extracted from Router Advertisements) used to delete entries that are no longer advertised

The Neighbor Cache contains information maintained by the Neighbor Unreachability Detection algorithm. A key piece of information is a neighbor's reachability state, which is one of five possible values:

- INCOMPLETE: Address resolution is in progress and the link-layer address of the neighbor has not yet been determined.

- REACHABLE: Roughly speaking, the neighbor is known to have been reachable recently (within tens of seconds ago).

- STALE: The neighbor is no longer known to be reachable but until traffic is sent to the neighbor, no attempt should be made to verify its reachability.

- DELAY: The neighbor is no longer known to be reachable, and traffic has recently been sent to the neighbor. Rather than probe the neighbor immediately, however, delay sending probes for a short while in order to give upper-layer protocols a chance to provide reachability confirmation.

- PROBE: The neighbor is no longer known to be reachable, and unicast Neighbor Solicitation probes are being sent to verify reachability.

Sending Algorithm

When sending a packet to a destination, a node uses a combination of the Destination Cache, the Prefix List, and the Default Router List to determine the IP address of the appropriate next hop, an operation known as "next-hop determination". Once the IP address of the next hop is known, the Neighbor Cache is consulted for link-layer information about that neighbor.

Next-hop determination for a given unicast destination operates as follows. The sender performs a longest prefix match against the Prefix List to determine whether the packet's destination is on- or off-link. If the destination is on-link, the next-hop address is the same as the packet's destination address. Otherwise, the sender selects a router from the Default Router List.

For efficiency reasons, next-hop determination is not performed on every packet that is sent. Instead, the results of next-hop determination computations are saved in the Destination Cache (which also contains updates learned from Redirect messages). When the sending node has a packet to send, it first examines the Destination Cache. If no entry exists for the destination, next-hop determination is invoked to create a Destination Cache entry.

Once the IP address of the next-hop node is known, the sender examines the Neighbor Cache for link-layer information about that neighbor. If no entry exists, the sender creates one, sets its state to INCOMPLETE, initiates Address Resolution, and then queues the data packet pending completion of address resolution. For multicast-capable interfaces Address Resolution consists of sending a Neighbor Solicitation message and waiting for a Neighbor Advertisement. When a Neighbor Advertisement response is received, the link-layer addresses is entered in the Neighbor Cache entry and the queued packet is transmitted.

For multicast packets, the next-hop is always the (multicast) destination address and is considered to be on-link.

Each time a Neighbor Cache entry is accessed while transmitting a unicast packet, the sender checks Neighbor Unreachability Detection related information according to the Neighbor Unreachability Detection algorithm. This unreachability check might result in the sender transmitting a unicast Neighbor Solicitation to verify that the neighbor is still reachable.

Next-hop determination is done the first time traffic is sent to a destination. As long as subsequent communication to that destination proceeds successfully, the Destination Cache entry continues to be used. If at some point communication ceases to proceed, as determined by the Neighbor Unreachability Detection algorithm, next-hop determination may need to be performed again.

IPv6 Stateless Address Autoconfiguration

The stateless address autoconfiguration mechanism [20] allows a host to generate its own addresses using a combination of locally available information and information advertised by routers. (Classic) routers advertise prefixes that identify the subnet(s) associated with a (classic) link, while hosts generate an "interface identifier" that uniquely identifies an interface on a subnet. An address is formed by combining the two. In the absence of routers, a host can only generate link-local addresses. However, link-local addresses are sufficient for allowing communication among nodes attached to the same link.

The stateless approach is used when a site is not particularly concerned with the exact addresses hosts use, so long as they are unique and properly routable. On the other hand, Dynamic Host Configuration Protocol for IPv6 (DHCPv6) [21] is used when a site requires tighter control over exact address assignments. Both stateless address autoconfiguration and DHCPv6 may be used simultaneously.

Each address has an associated lifetime that indicates how long the address is bound to an interface. When a lifetime expires, the binding (and address) become invalid and the address may be reassigned to another interface elsewhere in the Internet. To handle the expiration of address bindings gracefully, an address goes through two distinct phases while assigned to an interface. Initially, an address is "preferred", meaning that its use in arbitrary communication is unrestricted. Later, an address becomes "deprecated" in anticipation that its current interface binding will become invalid. While an address is in a deprecated state, its use is discouraged, but not strictly forbidden.

To ensure that all configured addresses are likely to be unique on a given link, nodes run a "duplicate address detection" algorithm on addresses before assigning them to an interface. The Duplicate Address Detection algorithm is performed on all addresses, independently of whether they are obtained via stateless autoconfiguration or DHCPv6.

Since host autoconfiguration uses information advertised by (classic) routers, routers need to be configured by some other means.

Nodes (both hosts and routers) begin the autoconfiguration process by generating a link-local address for the interface. A link-local address is formed by combining the well-known link-local prefix FE80::0 (of appropriate length) with an interface identifier as follows:

- The left-most 'prefix length' bits of the address are those of the link-local prefix
- The bits in the address to the right of the link-local prefix are set to all zeroes
- If the length of the interface identifier is N bits, the right- most N bits of the address are replaced by the interface identifier

A link-local address has an infinite preferred and valid lifetime; it is never timed out.

Before the link-local address can be assigned to an interface and used, however, a node must attempt to verify that this "tentative" address is not already in use by another node on the link. Specifically, it sends a Neighbor Solicitation message containing the tentative address as the target. If another node is already using that address, it will return a Neighbor Advertisement saying so.

If a node determines that its tentative link-local address is not unique, autoconfiguration stops and manual configuration of the interface is required. To simplify recovery in this case, it should be possible for an administrator to supply an alternate interface identifier that overrides the default identifier in such a way that the autoconfiguration mechanism can then be applied using the new (presumably unique) interface identifier. Alternatively, link-local and other addresses will need to be configured manually.

Once a node ascertains that its tentative link-local address is unique, it assigns the address to the interface. At this point, the node has IP-level connectivity with neighboring nodes. The remaining autoconfiguration steps are performed only by hosts.

The next phase of autoconfiguration involves obtaining a Router Advertisement or determining that no routers are present. If routers are present, they will send Router Advertisements that specify what sort of autoconfiguration a host can do.

Routers send Router Advertisements periodically, but the delay between successive advertisements will generally be longer than a host performing autoconfiguration will want to wait. To obtain an advertisement quickly, a host sends one or more Router Solicitations to the all-routers multicast group.

Router Advertisements also contain zero or more Prefix Information options that contain information used by stateless address autoconfiguration to generate global addresses. One Prefix Information option field, the "autonomous address-configuration flag", indicates whether or not the option even applies to stateless autoconfiguration. If it does, additional option fields contain a subnet prefix, together with lifetime values, indicating how long addresses created from the prefix remain preferred and valid.

1.15 ND over 6LoWPANs

IPv6 Neighbor Discovery provides several important mechanisms used for router discovery, address resolution, Duplicate Address Detection, and Redirect messages, along with prefix and parameter discovery. Following power-on and initialization of the network in IPv6 Ethernet networks, a node joins the solicited-node multicast address on the interface and then performs Duplicate Address Detection (DAD) for the acquired link-local address by sending a solicited-node multicast message to the link. After that, it sends multicast messages to the all-routers multicast address to solicit Router Advertisements (RAs). If the host receives a valid RA with the A (autonomous address configuration) flag, it autoconfigures the IPv6 address with the advertised prefix in the RA message. Besides this, the IPv6 routers usually send RAs periodically on the network. RAs are sent to the all-nodes multicast address. Nodes send Neighbor Solicitation/ Neighbor Advertisement messages to resolve the IPv6 address of the destination on the link. The Neighbor Solicitation messages used for address resolution are multicast. The Duplicate Address Detection procedure and the use of periodic Router Advertisement messages assume that the nodes are powered on and reachable most of the time.

In Neighbor Discovery, the routers find the hosts by assuming that a subnet prefix maps to one broadcast domain, and then they multicast Neighbor Solicitation messages to find the host and its link-layer address. Furthermore, the DAD use of multicast assumes that all hosts that autoconfigure IPv6 addresses from the same prefix can be reached using link-local multicast messages.

Note that the L (on-link) bit in the Prefix Information Option (PIO) of the RA can be set to zero in Neighbor Discovery, which makes the host not use multicast Neighbor Solicitation (NS) messages for address resolution of other hosts, but routers still use multicast NS messages to find the hosts.

Considering the characteristics previously illustrated in a LoWPAN, and the IPv6 Neighbor Discovery protocol design, some optimizations and extensions to Neighbor Discovery are useful for the wide deployment of IPv6 over low-power and lossy networks [22].

The optimized protocol defines three new ICMPv6 message options: the Address Registration Option (ARO), the Authoritative Border Router Option (ABRO), and the 6LoWPAN Context Option (6CO). It also defines two new ICMPv6 message types: the Duplicate Address Request (DAR) and the Duplicate Address Confirmation (DAC).

Definitions

- 6LoWPAN link: A link determined by single IP hop reachability of neighboring nodes. These are considered links with undetermined connectivity properties

- 6LoWPAN Node (6LN): A 6LoWPAN node is any host or router participating in a LoWPAN. This term is used when referring to situations in which either a host or router can play the role described

- 6LoWPAN Router (6LR): An intermediate router in the LoWPAN that is able to send and receive Router Advertisements (RAs) and Router Solicitations (RSs) as well as forward and route IPv6 packets. 6LoWPAN routers are present only in route-over topologies.

- 6LoWPAN Border Router (6LBR): A border router located at the junction of separate 6LoWPAN networks or between a 6LoWPAN network and another IP network. There may be one or more 6LBRs at the 6LoWPAN network boundary. A 6LBR is the responsible authority for IPv6 prefix propagation for the 6LoWPAN network it is serving. An isolated LoWPAN also contains a 6LBR in the network, which provides the prefix(es) for the isolated network

- Router: Either a 6LR or a 6LBR. Note that nothing precludes a node being a router on some interfaces and a host on other interfaces

- Mesh-under: A topology where nodes are connected to a 6LBR through a mesh using link-layer forwarding. Thus, in a mesh-under configuration, all IPv6 hosts in a LoWPAN are only one IP hop away from the 6LBR. This topology simulates the typical IP-subnet topology with one router with multiple nodes in the same subnet

- Route-over: A topology where hosts are connected to the 6LBR through the use of intermediate layer-3 (IP) routing. Here, hosts are typically multiple IP hops away from a 6LBR. The route-over topology typically consists of a 6LBR, a set of 6LRs, and hosts

- Non-transitive link: A link that exhibits asymmetric reachability, i.e. a link where non-reflexive and/or non-transitive reachability is part of normal operation. (Non- reflexive reachability means packets from A reach B, but packets from B don't reach A. Non-transitive reachability means packets from A reach B, and packets from B reach C, but packets from A don't reach C.) Many radio links exhibit these properties

- Header compression context: Address information shared across a LoWPAN and used by 6LoWPAN header compression to enable the elision of information that would otherwise be sent repeatedly. In a "context", a (potentially partial) address is associated with a Context Identifier (CID), which is then used in header compression as a shortcut for (parts of) a source or destination address

- Registration: The process during which a LoWPAN node sends a Neighbor Solicitation message with an Address Registration Option to a router creating a Neighbor Cache Entry (NCE) for the LoWPAN node with a specific timeout. Thus, for 6LoWPAN routers, the Neighbor Cache doesn't behave like a cache. Instead, it behaves as a registry of all the host addresses that are attached to the router

These Neighbor Discovery optimizations are applicable to both mesh-under and route-over configurations. In a mesh-under configuration, only

6LoWPAN Border Routers and hosts exist; there are no 6LoWPAN routers in mesh-under topologies. The most important part of the optimizations is the evolved host-to-router interaction that allows for sleeping nodes and avoids using multicast Neighbor Discovery messages except for the case of a host finding an initial set of default routers, and redoing such determination when that set of routers have become unreachable.

The protocol also provides for header compression by carrying header compression information in a new option in Router Advertisement messages.

In addition, there are separate mechanisms that can be used between 6LRs and 6LBRs to perform multihop Duplicate Address Detection and distribution of the prefix and compression context information from the 6LBRs to all the 6LRs, which in turn use normal Neighbor Discovery mechanisms to convey this information to the hosts.

The protocol is designed so that the host-to-router interaction is not affected by the configuration of the 6LoWPAN; the host-to-router interaction is the same in a mesh-under and route-over configuration.

The optimizations and extensions to IPv6 Neighbor Discovery protocol are the following:

- Host-initiated refresh of Router Advertisement information. This removes the need for periodic or unsolicited Router Advertisements from routers to hosts. However, if the routers use RAs to distribute prefix and/or context information across a route-over topology, that might require periodic RA messages
- No Duplicate Address Detection (DAD) is performed if EUI-64-based IPv6 addresses are used (as these addresses are assumed to be globally unique)
- DAD is optional if DHCPv6 is used to assign addresses
- A new address registration mechanism using a new Address Registration Option between hosts and routers. This removes the need for routers to use multicast Neighbor Solicitations to find hosts and supports sleeping hosts. This also enables the same IPv6 address prefix(es) to be used across a route-over 6LoWPAN. It provides the host-to-router interface for Duplicate Address Detection
- A new Router Advertisement option, the 6LoWPAN Context Option, for context information used by 6LoWPAN header compression. The

6LoWPAN Context Option (6CO) carries prefix information for LoWPAN header compression and is similar to the PIO. However, the prefixes can be remote as well as local to the LoWPAN, since header compression potentially applies to all IPv6 addresses. This option allows for the dissemination of multiple contexts identified by a CID. A context may be a prefix of any length or an address (/128), and up to 16 6COs may be carried in an RA message

- A new mechanism to perform Duplicate Address Detection across a route-over 6LoWPAN using the new Duplicate Address Request and Duplicate Address Confirmation messages. For the multihop DAD exchanges between a 6LR and 6LBR the protocol avoids reusing the NS and NA messages for this purpose, since these messages are not subject to the hop limit=255 check as they are forwarded by intermediate 6LRs

- New mechanisms to distribute prefixes and context information across a route-over network that uses a new Authoritative Border Router Option to control the flooding of configuration changes. ABRO is needed when RA messages are used to disseminate prefixes and context information across a route-over topology. In this case, 6LRs receive PIOs from other 6LRs. This implies that a 6LR can't just let the most recently received RA win. In order to be able to reliably add and remove prefixes from the 6LoWPAN, we need to carry information from the authoritative 6LBR. This is done by introducing a version number that the 6LBR sets and that 6LRs propagate as they propagate the prefix and context information with this ABRO. When there are multiple 6LBRs, they would have separate version number spaces. Thus, this option needs to carry the IP address of the 6LBR that originated that set of information. A 6LBR should always include an ABRO in the RAs it sends, listing itself as the 6LBR address

The 6LBRs are responsible for managing the prefix(es) assigned to the 6LoWPAN, using manual configuration, DHCPv6 Prefix Delegation or other mechanisms. In an isolated LoWPAN, a Unique Local Address (ULA) prefix should be generated by the 6LBR. If the LoWPAN has multiple 6LBRs, then they should be configured with the same set of prefixes. The set of prefixes is included in the RA messages.

Host-to-Router Interaction

A host sends Router Solicitation messages at startup and also when the Neighbor Unreachability Detection (NUD) of one of its default routers fails.

Hosts receive Router Advertisement messages typically containing the Authoritative Border Router Option (ABRO) and may optionally contain one or more 6LoWPAN Context Options (6COs) in addition to the existing Prefix Information Options (PIOs).

The routers need to know the set of host IP addresses that are directly reachable and their corresponding link-layer addresses. This needs to be maintained as the radio reachability changes. For this purpose, an Address Registration Option (ARO) is introduced, which can be included in unicast NS messages sent by hosts. Thus, it can be included in the unicast NS messages that a host sends as part of NUD to determine that it can still reach a default router. The same option is included in corresponding NA messages with a Status field indicating the success or failure of the registration. This option is always host initiated.

When a host has configured a non-link-local IPv6 address, it registers that address with one or more of its default routers using the ARO in an NS message. The host chooses a lifetime of the registration and repeats the ARO periodically (before the lifetime runs out) to maintain the registration.

As part of the optimizations, address resolution is not performed by multicasting Neighbor Solicitation messages. Instead, the routers maintain Neighbor Cache Entries for all registered IPv6 addresses. If the address is not in the Neighbor Cache in the router, then the address either doesn't exist, is assigned to a host attached to some other router in the 6LoWPAN, or is external to the 6LoWPAN. In a route-over configuration, the routing protocol is used to route such packets toward the destination. Hosts in a LoWPAN use the ARO in the NS messages they send as a way to maintain the Neighbor Cache in the routers, thereby removing the need for multicast NSs to do address resolution.

When the ARO is used by hosts, an SLLAO (Source Link-Layer Address Option) must be included, and the address that is to be registered must be the IPv6 source address of the NS message.

```
6LN                                                6LR

 |                                                  |
 |      ---------- Router Solicitation -------->    |
 |                      [SLLAO]                     |
 |                                                  |
 |      <-------- Router Advertisement ---------    |
 |               [PIO + 6CO + ABRO + SLLAO]         |
```

Figure 1.8: ND6: RS and RA exchange

```
6LN                          6LR                          6LBR

 |                            |                            |
 | --- NS with Address Reg -->|                            |
 |      [ARO + SLLAO]         |                            |
 |                            |                            |
 |                            | ----------- DAR ---------->|
 |                            |                            |
 |                            | <---------- DAC ----------- |
 |                            |                            |
 | <-- NA with Address Reg ---|                            |
 |      [ARO with Status]     |
```

Figure 1.9: ND6: NS and NA exchange, with registration

Router-to-Router Interaction

The new router-to-router interaction is only for the route-over configuration where 6LRs are present. 6LRs MUST act like a host during system startup and prefix configuration by sending Router Solicitation messages and autoconfiguring their IPv6 addresses, unlike routers in the non-6LoWPAN ND6. When multihop prefix and context dissemination are used, then the 6LRs store the ABRO, 6CO, and prefix information received (directly or indirectly) from the 6LBRs and redistribute this information in the Router Advertisement they send to other 6LRs or send to hosts in response to a Router Solicitation. There is a Version Number field in the ABRO, which is used to limit the flooding of updated information between the 6LRs.

A 6LR can perform Duplicate Address Detection against one or more 6LBRs using the new Duplicate Address Request (DAR) and Duplicate Address Confirmation (DAC) messages, which carry the information from the Address Registration Option. The DAR and DAC messages will be forwarded between the 6LR and 6LBRs; thus, the old rule for checking hop limit=255 does not apply to the DAR and DAC messages. Those multihop DAD messages MUST NOT modify any Neighbor Cache Entries on the routers, since we do not have the security benefits provided by the hop limit=255 check.

1.16 RPL

RPL [23] stands for "IPv6 Routing Protocol for Low-Power and Lossy Networks".

Low-power and Lossy Networks (LLNs) consist largely of constrained nodes (with limited processing power, memory, and sometimes energy when they are battery operated or energy scavenging) which are interconnected by lossy links, typically supporting only low data rates, that are usually unstable with relatively low packet delivery rates. Unlike traditional IP networks, LLNs must handle the lossy and dynamic nature of low-power link technologies (e.g. IEEE 802.15.4). To address these concerns, the IETF began to focus work on the use of IP in LLNs by forming the IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) working group to standardize the use of IPv6 in IEEE 802.15.4 networks. The IETF then formed the Routing over Low Power and Lossy Links (ROLL) group to specify the Routing Protocol for LLNs (RPL), i.e. an IP-layer routing protocol designed specifically for LLNs. RPL now is a proposed standard and is the most exploited in LLNs.

RPL supports the route-over routing mechanism.

RPL is a distance vector protocol: routers using distance-vector protocol do not have knowledge of the entire path to a destination. Distance-vector protocols are based on calculating the direction and distance to any link in a network: "Direction" usually means the next hop address and the exit interface, while "Distance" is a measure of the cost to reach a certain node. The least expensive route between any two nodes is the route with minimum distance. Each node maintains a vector (table) of minimum distance to every node.

RPL forms a tree-like topology also called Destination Oriented Directed Acyclic Graph (DODAG): each node in a RPL network has a preferred parent which acts like a gateway for that node. If a node does not have an entry in its routing table for a packet, the node simply forwards it to its preferred parent and so on until it either reaches the destination or a common parent which forwards it down the tree towards the destination. The nodes in a RPL network have routes for all the nodes down the tree. When a node instead wants to send a packet to the root, it simply sends the packet to its preferred parent in the tree, and the preferred parent then sends the packet to his preferred parent and so on until the packet reaches the root.

A RPL Instance contains one or more DODAG roots. These roots may operate independently, or they may coordinate over a network that is not necessarily as constrained as an LLN.

RPL builds the DODAG using the Objective Function (OF), illustrated in the next paragraph.

Objective Functions

The Objective Function (OF) uses routing metrics to form the DODAG based on some algorithm or calculation formula. Basically the objective functions optimize or constrain the routing metrics that are used to form the routes and hence help in choosing the best route. The OF defines how RPL nodes select and optimize routes within a RPL Instance. An OF defines how nodes translate one or more metrics and constraints into a

Rank, which approximates the node's distance from a DODAG root. For example, if an RPL instance uses an Objective Function that minimizes hop count, RPL will select paths with a minimum hop count. RPL requires that all nodes in a network use a common Objective Function.

The cost of reaching a destination for a node is calculated using various route metrics. LLN's metrics can be categorized as:

- Node metrics: e.g. node energy, hop count..
- Link metrics: e.g. throughput, latency, link quality level, ETX, link color..

The Expected Transmission Count (ETX) of a link is the expected number of transmissions required to send a packet over that link. A path's ETX is the sum of the ETX of all the links along the path: for instance, the ETX of a path with 3 links of 100 percent delivery ratio is 3, whereas the ETX of a path with 2 links of 50 percent delivery ratio is 4.

RPL Identifiers

Different terms need to be defined to fully understand RPL management:

- RPLInstanceID: identifies a set of one or more DODAGs. A network can have multiple RPLInstanceIDs, each one defines an independent set of DODAGs, optimized for different OFs and/or applications. The set of DODAGs identified by a RPLInstanceID is called a RPL Instance. All DODAGs in the same RPL Instance use the same Objective Function
- DODAGID: the scope of a DODAGID is a RPL Instance. The combination of RPLInstanceID and DODAGID uniquely identifies a single DODAG in the network. A RPL Instance may have multiple DODAGs, each one has a unique DODAGID
- DODAGVersionNumber: the scope of a DODAGVersionNumber is a DODAG. A DODAG is sometimes reconstructed from the DODAG root, by incrementing the DODAGVersionNumber. The combination of RPLInstanceID, DODAGID, and DODAGVersionNumber uniquely identifies a DODAG Version

- Rank: the scope of Rank is a DODAG Version. Rank establishes a partial order over a DODAG Version, defining individual node positions with respect to the DODAG root

A RPL Instance may comprise:

- a single DODAG with a single root: for example, a DODAG optimized to minimize latency rooted at a single centralized lighting controller in a Home Automation application
- multiple uncoordinated DODAGs with independent roots (differing DODAGIDs): for example, multiple data collection points in an urban data collection application that do not have suitable connectivity to coordinate with each other or that use the formation of multiple DODAGs as a means to dynamically and autonomously partition the network
- a single DODAG with a virtual root that coordinates LLN sinks (with the same DODAGID) over a backbone network: for example, multiple border routers operating with a reliable transit link, e.g., in support of an IPv6 Low-Power Wireless Personal Area Network (6LoWPAN) application, that are capable of acting as logically equivalent interfaces to the sink of the same DODAG
- a combination of the above as suited to some application scenario

Each RPL packet is associated with a particular RPLInstanceID.

Figure 1.10 on the next page depicts an example of a RPL Instance comprising three DODAGs with DODAG roots R1, R2, and R3. Each of these DODAG roots advertises the same RPLInstanceID. The lines depict connectivity between parents and children.

ICMPv6 RPL Control Messages

RPL uses three types of control messages for creating and maintaining RPL topology and routing table: DODAG Information Object (DIO), DODAG Information Solicitation (DIS) and DODAG Destination Advertisement Object (DAO). These messages are new types of ICMPv6 messages with different purposes and structured as follow:

```
+----------------------------------------------------------------+
|                                                                |
| +--------------+                                               |
| |              |                                               |
| |    (R1)      |          (R2)                  (R3)           |
| |    /  \      |          /| \                 / |  \          |
| |   /    \     |         / |  \               /  |   \         |
| | (A)    (B)   |      (C)  |  (D)     ...   (F) (G)  (H)       |
| | /|\     |\   |      /    | / |\           |\  |     |        |
| | : : :   : :  |     :    (E)  : :          :  `:     :        |
| |              |          / \                                  |
| +--------------+         :   :                                 |
|     DODAG                                                      |
|                                                                |
+----------------------------------------------------------------+
                         RPL Instance
```

Figure 1.10: RPL instance example

- DODAG Information Object (DIO): carries information that allows a node to discover a RPL Instance, learn its configuration parameters, select a DODAG parent set and maintain the DODAG. When a RPL network starts, the nodes start exchanging the information about the DODAG using DIO messages which contains information about the DODAG configuration and help the nodes to join the DODAG and select parents. The format of the DIO is illustrated in figure 1.11 on the facing page:

  The DIO message may carry valid options. Between these options, there are the DAG Metric Container Option and the Prefix Information Option. The metric that MRHOF uses is determined by the metrics in the DIO Metric Container Option. In the absence of a metric in the DIO Metric Container, MRHOF defaults to using ETX to compute Rank. A RPL node may use the Prefix Information Option (PIO) for the purpose of Stateless Address Autoconfiguration (SLAAC) from a prefix advertised by a parent, and to advertise its own address. The prefix information exchange in RPL is carried by the DIO messages, thus there's no need to exploit RS and RA messages (in Contiki it's possible to disable the sending of RS and RA via the `UIP_ND6_SEND_RA` label). It is not correct to think at RPL and ND6 as two conflicting protocols: in fact, NS and NA messages are still useful exploited in parallel with RPL as they report the connectivity status between two devices

- DODAG Information Solicitations (DIS): used by any node to explic-

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| RPLInstanceID |Version Number |             Rank              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|G|0| MOP | Prf |     DTSN      |     Flags     |   Reserved    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                            DODAGID                            +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Option(s)...
+-+-+-+-+-+-+-+-+
```

Figure 1.11: RPL DIO message

itly solicit the DIO messages from the neighbor nodes. It is triggered by the node in case it could not receive a DIO after a predefined time interval. Its use is analogous to that of a Router Solicitation as specified in IPv6 Neighbor Discovery

- Destination Advertisement Object (DAO): used to propagate destination information Upward along the DODAG and to maintain Downward routes. How RPL constructs and maintains Downward routes is explained later. The DAO RPL Target option is used to indicate a target IPv6 address, prefix, or multicast group that is reachable or queried along the DODAG. In Contiki 3.0, RPL target option contains IPv6 addresses and not prefixes

RPL control messages have the scope of a link. The source address of RPL control messages is a link-local address, and the destination address is either the all-RPL-nodes multicast address or a link-local unicast address of the destination. The all-RPL-nodes multicast address is a new address with a value of ff02::1a. the RPL Control Message consists of an ICMPv6 header followed by a message body. The message body is comprised of a message base and possibly a number of options as illustrated in Figure 1.12 on the next page.

The RPL control message is an ICMPv6 information message with a Type of 155.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
.                             Base                              .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
.                           Option(s)                           .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1.12: RPL Control message

The Code field identifies the type of RPL control message:

- 0x00: DODAG Information Solicitation
- 0x01: DODAG Information Object
- 0x02: Destination Advertisement Object
- 0x03: Destination Advertisement Object Acknowledgment

A high-level overview of the distributed algorithm, which constructs the DODAG, is as follows:

- Some nodes are configured to be DODAG roots, with associated DODAG configurations
- Nodes advertise their presence, affiliation with a DODAG, routing cost, and related metrics by sending link-local multicast DIO messages to all-RPL-nodes
- Nodes listen for DIOs and use their information to join a new DODAG (thus, selecting DODAG parents), or to maintain an existing DODAG, according to the specified Objective Function and Rank of their neighbors
- Nodes provision routing table entries, for the destinations specified by the DIO message, via their DODAG parents in the DODAG Version. Nodes that decide to join a DODAG can provision one or more DODAG parents as the next hop for the default route and a number of other external routes for the associated instance

Downward Routes

RPL constructs and maintains Downward routes with DAO messages. There are two modes a RPL Instance may choose from for maintaining Downward routes.

In the first mode, called "Storing", nodes store Downward routing tables for their sub-DODAG. Each hop on a Downward route in a storing network examines its routing table to decide on the next hop. In the second mode, called "Non-Storing", nodes do not store Downward routing tables. Contiki supports only the Storing mode.

To establish Downward routes, RPL nodes send DAO messages Upward. The next-hop destinations of these DAO messages are called "DAO parents". The collection of a node's DAO parents is called the "DAO parent set".

In Storing mode, the DAO message is unicast by the child to the selected parent(s). In Non-Storing mode, the DAO message is unicast to the DODAG root. The DAO message may optionally, upon explicit request or error, be acknowledged by its destination with a Destination Advertisement Acknowledgement (DAO-ACK) message back to the sender of the DAO.

Destination Advertisement may be configured to be entirely disabled, or operate in either a Storing or Non-Storing mode, as reported in the MOP in the DIO message.

- All nodes who join a DODAG must abide by the MOP setting from the root. Nodes that do not have the capability to fully participate as a router, e.g., that do not match the advertised MOP, may join the DODAG as a leaf

- If the MOP is 0, indicating no Downward routing, nodes must not transmit DAO messages and may ignore DAO messages

- In Non-Storing mode, the DODAG root should store source routing table entries for destinations learned from DAOs. The DODAG root must be able to generate source routes for those destinations learned from DAOs that were stored

- In Storing mode, all non-root, non-leaf nodes must store routing table entries for destinations learned from DAOs

1.17 Power-Line Communication (PLC)

Power-line communication (PLC) [24] is a communication technology that enables the transfer of data over existing power line cables. Like any communication technology, PLC, works as follows: a sender modulates the data to be sent, injects it onto the medium (in this case power lines), and the receiver de-modulates (like a modem) the data to read it.

Unlike other communication technologies like Ethernet or optical fiber, PLC does not need extra wiring, but re-uses existing wiring; this means that theoretically any line-powered appliance can be part of a data network.

Power-Line Communication solutions are divided in two big categories: narrowband and broadband.

Functionally, there are two basic forms for both narrowband and broadband PLC [25] :

- Narrowband in-house applications: where household wiring is used for low bit rate services like home automation and intercoms
- Narrowband outdoor applications. These are mainly used by the utility companies for automatic meter reading and remote surveillance and control
- Broadband In-house mains power wiring can be used for high speed data transmission for home networking
- Broadband over Power Line: outdoor mains power wiring can be used to offer broadband internet access

1.17.1 Narrowband PLC

Narrowband PLC works at low frequencies (3–500 kHz), low data rates (up to 100s of kbps), and has long range (up to several kilometers), which can be extended using repeaters. It can be applied in the Smart Grid and for smart energy generation (for instance, micro-inverters for solar panels) or in Smart Home/Home Control applications.

Typically home-control power-line communication devices operate by modulating in a carrier wave of between 20 and 200 kHz into the household wiring at the transmitter. The carrier is modulated by digital signals. Each receiver in the system has an address and can be individually commanded by the signals transmitted over the household wiring and decoded at the receiver. These devices may be either plugged into regular power outlets, or permanently wired in place. Since the carrier signal may propagate to nearby homes (or apartments) on the same distribution system, these control schemes have a "house address" that designates the owner.

Outdoor Protocols

There are two main protocols in narrowband PLC outdoor application: G3 and PRIME.

G3 In 2011, several companies including distribution network operators, meter vendors and chip vendors (Maxim Integrated, Texas Instruments, STMicroelectronics) founded the G3-PLC Alliance to promote G3-PLC technology. G3-PLC, or 3rd Generation Power Line Communication, is the low layer protocol to enable large scale infrastructure on the electrical grid. The G3-PLC is a plug-and-play solution that uses the existing electric networks to carry information, so installation efforts are minimal. It is a radio-free solution that allows consumers to effectively monitor and manage their electricity consumption. With the ability to cross transformers, infrastructure costs are reduced and with its support of IPv6, G3-PLC will support powerline communications into the future. Two-way communications networks based on G3-PLC will provide electricity network operators with intelligent monitoring and control capabilities. Operators will be able to monitor electricity consumption throughout the grid in real time, implement variable tariff schedules, and set limits on electricity consumption to better manage peak loads.

In turn, consumers will have real-time visibility into their electricity consumption, thus promoting demand-side conservation. With the addition of variable tariff schedules, users will be encouraged to reduce electricity consumption during peak usage times.

Ultimately, intelligent network management techniques provide a smarter solution for the environment. Rather than build more power plants to support worst-case scenarios, network operators will be able to optimally utilize existing resources. At the same time, demand-side management will function as a form of indirect generation by better balancing the distribution of loads.

G3-PLC may operate on CENELEC A band (35 kHz to 91 kHz) or CENELEC B band (98 kHz to 122 kHz)

PRIME PRIME is an acronym for "PoweRline Intelligent Metering Evolution" and was conceived in 2007. Most popular usage of PRIME is in Advanced Metering Infrastructures (AMI), which are systems that measure, collect, and analyze energy usage, and communicate with metering devices such as electricity meters, gas meters, heat meters, and water meters, either on request or on a schedule. These systems include hardware, software, communications, consumer energy displays and controllers, customer associated systems, Meter Data Management (MDM) software, and supplier business systems. Government agencies and utilities are turning toward advanced metering infrastructure (AMI) systems as part of larger “Smart Grid” initiatives. AMI extends current advanced meter reading (AMR) technology by providing two way meter communications, allowing commands to be sent toward the home for multiple purposes, including “time-of-use” pricing information, demand-response actions, or remote service disconnects.

The PRIME specification [26] is structured into Physical Layer, MAC Layer and Convergence Layer. Since specification version 1.4, PRIME exploits high frequencies up to 471 kHz. Thus, raw data rates are eight times as high as in CENELEC A band.

In a PRIME subnetwork two device types exist: Base nodes and Service nodes. Base nodes manage subnetwork resources and connections. All devices, which are not Base nodes are Service nodes. Service nodes register with Base nodes to become part of a subnetwork.

The topology generated by a PRIME subnetwork is a tree with the Base node as trunk. To extend the subnetwork range, a Base node can promote a Service node from terminal state to switch state. Switches relay data in the subnetwork and build the branch points of the tree.

Powerline is a shared communication media. Base nodes and switches announce their presence with beacon messages in well specified intervals. These beacons provide a common time notion to a subnetwork. Time is split into shared contention period (SCP) and contention free period (CFP). During SCP, nodes can access the channel using CSMA/CA. For the CFP period, the base node arbitrates channel access.

To reduce transmission overhead, PRIME uses dynamic addresses to address nodes in a subnetwork. The addressing scheme resembles the tree structure of the subnetwork and consists of local switch id, local node id and local connection id. Routes are established during service node registration. PRIME makes use of address structure for packet routing, which reduces state information needed by service nodes. Base node and service nodes monitor network attachment based on periodic exchanged control messages, so called "keep alive messages".

PRIME allows connection oriented communication. The PRIME MAC layer includes control mechanism/messages to open and close unicast, multicast and broadcast connections.

PRIME specifies a security profile for encryption of MAC layer packets. Encryption is based on AES-CCM with 128bit keys and key derivation mechanism recommended by NIST.

In-house Protocols

Main powerline narrowband in-house protocols are [27]:

- universal powerline bus, introduced in 1999, uses pulse-position modulation
- X10
- LonTalk, part of the LonWorks home automation product line, was accepted as part of some automation standards

1.17.2 Broadband PLC

Broadband PLC works at higher frequencies (1.8-250 MHz), high data rates (up to 100s of Mbps) and is used in short-range applications. High frequency communication may (re)use large portions of the radio spectrum for communication, or may use select (narrow) band(s), depending on the technology.

Broadband over power line (BPL) is a system to transmit two-way data over existing AC MV (medium voltage) electrical distribution wiring, between transformers, and AC LV (low voltage) wiring between transformer and customer outlets (typically 110 to 240V). This avoids the expense of a dedicated network of wires for data communication, and the expense of maintaining a dedicated network of antennas, radios and routers in wireless network.

In-house Protocols

Power line networking is being developed by the HomePlug Powerline Alliance and there are two main standards [28]: HomePlug 1.0 and HomePlug AV.

- HomePlug 1.0 was first introduced back in 2001 and has a cap speed of 14Mbps. It's now becoming obsolete
- HomePlug AV, introduced in 2005, has an initial cap speed of 200Mbps, which is fast enough to carry multimedia content, hence the AV designation for Audio and Video. This standard also supports 128-bit AES encryption for security. HomePlug AV is backward-compatible with HomePlug 1.0 and is marketed as Powerline AV (or Powerline AV 200)
- Powerline AV adapters have a real cap of just 100Mbps as they also support the regular 10/100 Ethernet standard. In testing, the actual sustained speed of these adapters is somewhere from 20Mbps to 60Mbps

HomePlug AV got a boost with the ratification of the IEEE 1901 specification in 2010. Whereas previously Powerline was an independent standard, this brought it under the same umbrella as the other networking standards and protocols. This specification guarantees interoperability between adapters from different vendors, and on top of that the cap speed is now increased to 500Mbps. This much faster HomePlug AV is marketed as Powerline AV 500.

Powerline AV 500 offers real-life cap speeds of either 100Mbps or 500Mbps depending on the type of network port the adapter device supports, be it regular 10/100 Ethernet or Gigabit Ethernet. In real-world testing, Powerline AV 500 indeed offers significantly higher sustained speed than Powerline AV, giving speeds ranging from 90Mbps to 200Mbps.

Chapter 2

State of the Art

2.1 Hardware

Internet of things is made by sets of physical objects that use network support to exchange data. These objects can be sensors, software, boards and so on. From the hardware point of view, the market offers a lot of platforms that allow people to build their own IoT projects. The most known boards for building IoT projects are Arduino (with its several versions) and Raspberry. Even if in this thesis neither of these two boards are used, it's important to understand the difference between these two types of devices in order to understand the choice of the platform exploited in this work.

Arduino VS Raspberry

A Raspberry Pi, from different points of view, is similar to a computer, with a dedicated processor, memory, and a graphics driver for output through HDMI. It even runs a specially designed version of the Linux operating system. That makes it easy to install most Linux software, and allows to use the Pi as a functioning media streamer or video game emulator with a bit of effort.

Arduino boards are instead powered by a micro-controller, and are very simpler than a computer. They don't run a full operating system, but only execute a simpler firmware. Even if it's not possible to access to the basic functions that an operating system provides, on the other hand directly executing simple code is easier, and is accomplished with no operating system overhead. The main purpose of the Arduino board is to interface with sensors and devices, so it's great for hardware projects in which things have to respond to various sensor readings and manual input. It's great for interfacing with other devices and actuators, where a full operating system would be overkill for handling simple read and response actions.

Arduino is best suited when the main task is reading sensor data and changing values on motors or other devices [29]. Given the Arduino's low power requirements and upkeep, it's also a good choice if the device will be constantly running and requires little to no interaction.

Raspberry Pi is best suited when a certain task can not be handled by a microcontroller. The Pi makes a slew of operations easier to manage, whether you intend to connect to the Internet to read and write data, view media of any kind, or connect to an external display.

A rough comparison between the main boards is reported in table 2.1 on the facing page.

Microcontrollers

Main difference between IoT microcontrollers is the architecture number of bits, i.e. 8-bit, 16-bit and 32 bit architectures. There is no doubt that a 32-bit MCU has a higher performance capability than an 8-bit device, but the engineer is faced with the traditional decision of choosing between what is the best device available in the market against what the application actually needs.

Main IoT microcontrollers characteristics are reported in table 2.2 on page 84:

2.2 Software

Two WSN-oriented operating systems clearly stand out in the IoT domain: TinyOS and Contiki. Both are open-source projects, providing free systems to the WSN and more generally embedded devices oriented community of scientists and developers. This work utilizes the latter as software platform, since it is written in standard C language (whereas TinyOS uses a specific derivative named nesC), more well structured,

Table 2.1: main boards comparison

	Arduino UNO	**Arduino DUE**	**Raspberry Pi 2 B**	**Raspberry Pi 3 B**
€	22,05	40,98	42,97	46,98
Architecture	8-bit	32-bit	32-bit	64-bit
Processor	ATmega 328	SAM3X8E ARM Cortex-M3	Quad-core ARM cortex -A7	Quad-core ARM Cortex -A53
Processor Speed	16 MHz	84 MHz	900 MHz	1,2 GHz
Pins	20	66	40	40
Memory	SRAM 2KB, EEPROM 1KB	SRAM 96KB	1 GB RAM microSD card slot	1 GB RAM, microSD card slot
Ethernet	N/A	N/A	10/100	10/100
Bluetooth	N/A	N/A	N/A	BLE and 4,1 classic
Wireless	N/A	N/A	N/A	802.11n
Video	N/A	N/A	HDMI, Camera Serial Interface (CSI), Display Serial Interface (DSI), GPU Broadcom VideoCore IV	HDMI, Camera Serial Interface (CSI), Display Serial Interface (DSI), GPU Broadcom VideoCore IV
Audio	N/A	N/A	HDMI, 3.5mm analogue audio-video jack	HDMI, 3.5mm analogue audio-video jack
Supply voltage	5V	3,3V	5V	5,1V
Current cons (active mode)	200 uA/MHz	6 mA/MHz	4*20 mA	4*34,31 mA
Power cons	1 mW/MHz	19,8 mW/MHz	4*100 mW	4*175 mW

Table 2.2: main IoT microcontrollers features

	8-bit architecture		**32-bit architecture**	
	Min	Max	Min	Max
Clock Speed	10 Megaherz	24 Megaherz	50 Megaherz	216 Megaherz
RAM	64 Bytes	6 Kilobytes	8 Kilobytes	512 Kilobytes
EEPROM	64 Bytes	2 Kilobytes	-	-
Flash	1 Kilobytes	128 Kilobytes	16 Kilobytes	2 Megabytes
I/O pins	6	70	28	216
Supply voltage	1,8 V	5,5 V	1,7 V	3,6 V
Current cons	5 uA/MHz	500 uA/MHz	1,7 uA/MHz	472 uA/MHz
Power cons	9 uW/MHz	2,75 mW/MHz	0,51 uW/MHz	1,7 mW/MHz

and is more actively used and supported by both scientific and industrial communities.

2.2.1 Contiki 3.X OS

Contiki [30] is an open source, highly portable, multi-tasking operating system for memory-efficient networked embedded systems and wireless sensor networks.

Contiki is maintained by a group of developers from industry and academia lead by Adam Dunkels from the Swedish Institute of Computer Science. The Contiki team currently consists of sixteen developers from SICS, SAP AG, Cisco, Atmel, NewAE and TU Munich.

The Contiki operating system includes a network simulator called Cooja that allows to simulate networks of Contiki nodes, which may belong to either of the following three classes:

- emulated nodes: the entire hardware of each node is emulated
- Cooja nodes: the Contiki code for the node is compiled for and executed on the simulation host
- Java nodes: the behavior of the node must be reimplemented as a Java class

A single Cooja simulation may contain a mixture of nodes from either of the three classes. Emulated nodes can also be used to include non-Contiki nodes in a simulated network.

Another Contiki's utility that is exploited in this work is Tunslip. Tunslip is a tool used to bridge IP traffic between a host and another network element, typically a border router, over a serial line. Tunslip creates a virtual network interface (tun) on the host side and uses SLIP (serial line internet protocol) to encapsulate and pass IP traffic to and from the other side of the serial line. The network element sitting on the other side of the line does a similar job with it's network interface. The tun interface can be used like any real network interface: routing, traffic forwarding, Wireshark analysis, etc.

Current major version of Contiki is 3.0, released the 25th August 2015.

Tree Structure

Contiki tree structure holds the following main folders:

- apps: contains some ready-to-use applications
- core: contains the kernel code
- cpu: contains the CPUs configuration source code of the CPUs supported by Contiki
- dev: contains some devices implementation source code, as radio drivers, spi driver and other
- doc: contains the Contiki official documentation
- examples: contains some ready-to-use simple applications
- platform: contains the platforms configuration source code of the platforms supported by Contiki
- regression-tests: contains some regression tests to run in Cooja
- tools: contains some tools, among which can be found Tunslip and Cooja

Protothreads

Contiki is designed to run on classes of hardware devices that are severely constrained in terms of memory, power, processing power, and communication bandwidth. A typical Contiki system has memory on the order of kilobytes, a power budget on the order of milliwatts, processing speed measured in megahertz, and communication bandwidth on the order of hundreds of kilobits/second. This class of systems includes both various types of embedded systems as well as a number of old 8-bit computers.

To run efficiently on memory-constrained systems, the Contiki programming model is based on protothreads. A protothread [31, 32] is a memory-efficient programming abstraction that shares features of both multi-threading and event-driven programming to attain a low memory overhead of each protothread. Event-driven programming is a common programming model for memory-constrained embedded systems, including sensor networks. Compared to multi-threaded systems, event-driven systems do not need to allocate memory for per-thread stacks, which leads to lower memory requirements. For this reason, many operating systems for sensor networks, including TinyOS and Contiki, are based on an event-driven model.

An event-driven model does not support a blocking wait abstraction. Therefore, programmers of such systems frequently need to use state machines to implement control flow for high-level logic that cannot be expressed as a single event handler. Unlike state machines that are part of a system specification, the control-flow state machines typically have no formal specification, but are created on-the-fly by the programmer.

In the Contiki operating system, processes are implemented as protothreads running on top of the event-driven Contiki kernel. A process' protothread is invoked whenever the process receives an event. The event may be a message from another process, a timer event, a notification of sensor input, or any other type of event in the system. Processes may wait for incoming events using the protothread conditional blocking statements. Protothreads can be seen as blocking event handlers in that protothreads can run on top of an existing event-based kernel, without modifications to the underlying event-driven system. Protothreads running on top of an event-driven system can use the `PT_WAIT_UNTIL()` statement to block conditionally. The underlying event dispatching system does not need to know whether the event handler is a protothread or a regular event handler.

Source Code 2.1: the radio sleep cycle implemented with protothreads in pseudocode

```
radio_wake_protothread:
  PT_BEGIN
  while(true)
    radio_on()
    timer <- t_awake
    PT_WAIT_UNTIL(expired(timer))
    timer <- t_sleep
    if(not communication_complete())
      wait_timer <- t_wait_max
      PT_WAIT_UNTIL(communication_complete() or
                    expired(wait_timer))
    radio_off()
    PT_WAIT_UNTIL(expired(timer))
PT_END
```

Contiki Network Stack

Contiki provides three network stacks: the uIP [33] TCP/IP stack which provides IPv4 networking, the uIPv6 stack (developed by and contributed to Contiki by Cisco) for IPv6 networking, and the Rime stack which is a set of custom lightweight networking protocols designed specifically for low-power wireless networks. Rime stack is used mainly when IP overhead is too prohibitive. Rime is not deepen in this work as it is not exploited.

The uIP TCP/IP stack is intended to make it possible to communicate using the TCP/IP protocol suite even on small 8-bit micro-controllers. The uIP implementation is designed to have only the absolute minimal set of features needed for a full TCP/IP stack. It can only handle a single network interface and contains the IP, ICMP, UDP and TCP protocols. uIP is written in the C programming language.

The uIP stack can be run either as a task in a multitasking system, or as the main program in a single task system. In both cases, the main control loop, which is named “`tcpip_process`” and can be found in the core/net/tcpip.c source file, does two things repeatedly:

- Check if a packet has arrived from the network
- Check if a periodic timeout has occurred

If a packet has arrived, the input handler function, `uip_input()`, should be invoked by the main control loop. The input handler function will never block, but will return at once. When it returns, the stack or the application for which the incoming packet was intended may have produced one or more reply packets which should be sent out. If so, the network device driver should be called to send out these packets.

Periodic timeouts are used to drive TCP mechanisms that depend on timers, such as delayed acknowledgments, retransmissions and round-trip time estimations. When the main control loop infers that the periodic timer should fire, it should invoke the timer handler function `uip_periodic()`. Because the TCP/IP stack may perform retransmissions when dealing with a timer event, the network device driver should be called to send out the packets that may have been produced.

The uIP stack does not use explicit dynamic memory allocation. Instead, it uses two global buffers for holding packets and has a fixed table for holding connection state. The global packet buffers are large enough to contain one packet of maximum size. When a packet arrives from the network, the device driver places it in one global buffer (which is called "packetbuf") and calls the TCP/IP stack. If the packet contains data, the TCP/IP stack will notify the corresponding application. Because the data in the buffer will be overwritten by the next incoming packet, the application will either have to act immediately on the data or copy the data into a secondary buffer for later processing. The packet buffer will not be overwritten by new packets before the application has processed the data. Packets that arrive when the application is processing the data must be queued, either by the network device or by the device driver. If the buffers are full, the incoming packet is dropped.

The uIP stack is highly customizable: in fact, Contiki provides a lot of C macros that can be configured in order to achieve different objectives of power consumption, routing, performance etc. The Contiki build system is designed to make it easy to compile Contiki applications for either an hardware platform or for the Cooja simulation platform by simply supplying a different target parameter to the make command, without having to edit makefiles or modify the application code. For instance, to compile the application and a Contiki system for the ESB platform the command "make TARGET=esb" is used.

Main Contiki networking protocols and components are illustrated in figure 2.1 on the facing page, where the Network stack ("Netstack") is included in the yellow box. Instead, all the blue boxes are abstraction layers which interact between each others by exploiting the respective API: all layers in fact are conceived as drivers that hold several functions as "init", "`packet_input`", "`send_packet`", "on", "off" and so on.

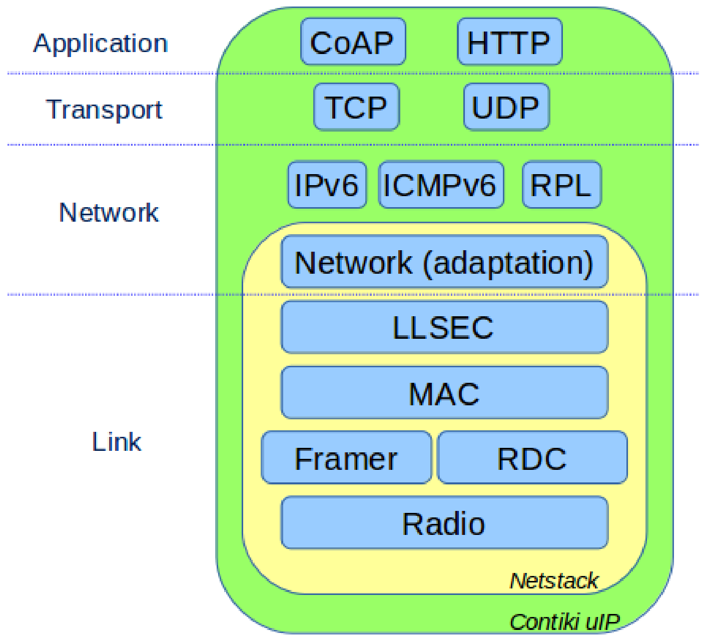


Figure 2.1: Contiki Netstack

For instance, the MAC driver is defined as follow:

Source Code 2.2: MAC driver

```c
struct mac_driver {
  char *name;

  /** Initialize the MAC driver */
  void (* init)(void);

  /** Send a packet from the Rime buffer  */
  void (* send)(mac_callback_t sent_callback, void *ptr);

  /** Callback for getting notified of incoming packet. */
  void (* input)(void);

  /** Turn the MAC layer on. */
  int (* on)(void);

  /** Turn the MAC layer off. */
  int (* off)(int keep_radio_on);

  /** Returns the channel check interval, expressed in
      clock_time_t ticks. */
  unsigned short (* channel_check_interval)(void);
};
```

CoAP

Contiki implements the IETF CoAP [34]: it is designed to provide a REST-like interface, but with a lower cost in terms of bandwidth and implementation complexity than HTTP-based REST interfaces. The Representational State Transfer (REST)[35] style is an abstraction of the architectural elements within a distributed hypermedia system; Perry and Wolf [36] distinguish three classes of architectural elements: processing elements (a.k.a. components), data elements and connecting elements (a.k.a. connectors); REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.

Network Drivers

There are two network drivers in Contiki: 6LoWPAN and Rime. 6LoWPAN is discussed in the next paragraphs, Rime is not discussed.

The adaptation layer, i.e. the layer which holds the network driver and the Radio layer are the only ones which can not be implemented by the “null protocols”, i.e. the protocols which hold functions which do nothing but invoke the adjacent layers.

LLSEC Drivers

The LLSec is the abbreviation for MAC Link Layer Security. The purpose of a Link Layer Security protocol is to ensure specific security properties of link layer PDUs, that is the PDUs of the protocol layer carrying the PDUs of the network layer (e.g. IP).

Encryption and security support was always sorely lacking in the previous versions of Contiki. Contiki 3.0 brings support for 802.15.4 link layer encryption with AES128 with CCM (Counter with CBC-MAC) mode, which is an encryption algorithm designed to provide both authentication and confidentiality. In addition, Contiki offers anti-replay protection, that assures the detection of duplicate information.

MAC Drivers

Contiki provides two MAC drivers, CSMA and NullMAC. CSMA is the default mechanism. The MAC layer receives incoming packets from the RDC layer and uses the RDC layer to transmit packets. If the RDC layer or the radio layer detects a radio collision, the MAC layer may retransmit the packet at a later point in time. The CSMA mechanism is currently the only MAC layer that retransmits packets if a collision is detected.

RDC Drivers

The Radio Duty-Cycle driver controls how the radio transceiver is turned on and off during the duty cycles, thus implementing the energy-saving strategy.

In the Contiki Netstack there is a distinction between the RDC layer and the MAC layer. This separation is done probably only in Contiki OS, which leaves the MAC layer only tasked with ordering and sequencing packet transmissions. Most modern MAC protocols do manage both of these aspects: as an example, the ContikiMAC protocol is itself implemented as a sole RDC driver. Furthermore, consider the CSMA protocol, which stands for “Carrier-Sense Medium Access”: the implementation of CSMA protocol in ContikiOS does not rely on the carrier sensing because the medium access is performed by RDC protocol.

Contiki has several RDC drivers. The most commonly used are ContikiMAC, X-MAC, CX-MAC, LPP, and NullRDC. ContikiMAC is the default mechanism that provides a very good power efficiency but is somewhat tailored for the 802.15.4 radio and the CC2420 radio transceiver. X-MAC is an older mechanism that does not provide the same power-efficiency as ContikiMAC but has less stringent timing requirements. CX-MAC (Compatibility X-MAC) is an implementation of X-MAC that has more relaxed timing than the default X-MAC and therefore works on a broader set of radios. LPP (Low-Power Probing) as a receiver-initiated RDC protocol. NullRDC is a "null" RDC layer that never switches the radio off and that therefore can be used for testing or for comparison with the other RDC drivers.

Framer

A MAC framer module is responsible for constructing and parsing the header in MAC frames. It is called directly by the RDC, and it converts link-layer headers to packet attributes when a packet arrives and vice-versa during a packet output. Actually, the framer is not categorized as a driver: in fact, it can be considered as a support module for the RDC, not as a concrete Netstack layer which interacts with the other layers.

Global Buffers

The packetbuf buffer is the global buffer which is shared among all the Netstack layers. When a packet arrives, it is placed from the Radio driver in the packetbuf. Radio driver is also responsible to deposit some packet meta data as RSSI (received signal strength indicator, is a measurement of the power present in a received radio signal) in some packet attributes. The second global buffer, which is called "`uip_buf`", is handled by the protocols which does not belong to the Netstack, except for the adaptation layer: in fact, it fills up the `uip_buf` buffer during a packet input process by processing the packetbuf information, and during a packet output process it exploits the information contained in the `uip_buf` in order to fill up the packetbuf buffer.

There is also another buffer which is important to be mentioned, but it's only implemented when using CSMA protocol: the packet queue buffer, which is called "queuebuf". The queuebuf is organized per neighbor: every neighbor holds a packet queue, which is handled by the CSMA driver.

Layers Interaction

An interesting example that illustrates how the layers interact between them could be the receive-and-reply-to-a-packet situation, illustrated in picture 2.2 on the next page

When the Radio driver wants to call the RDC driver in order to leave him the duty of handling the packet received in input (that he has already put in the packetbuf), he must exploit the RDC API in the following way:

```
NETSTACK_RDC.input()
```

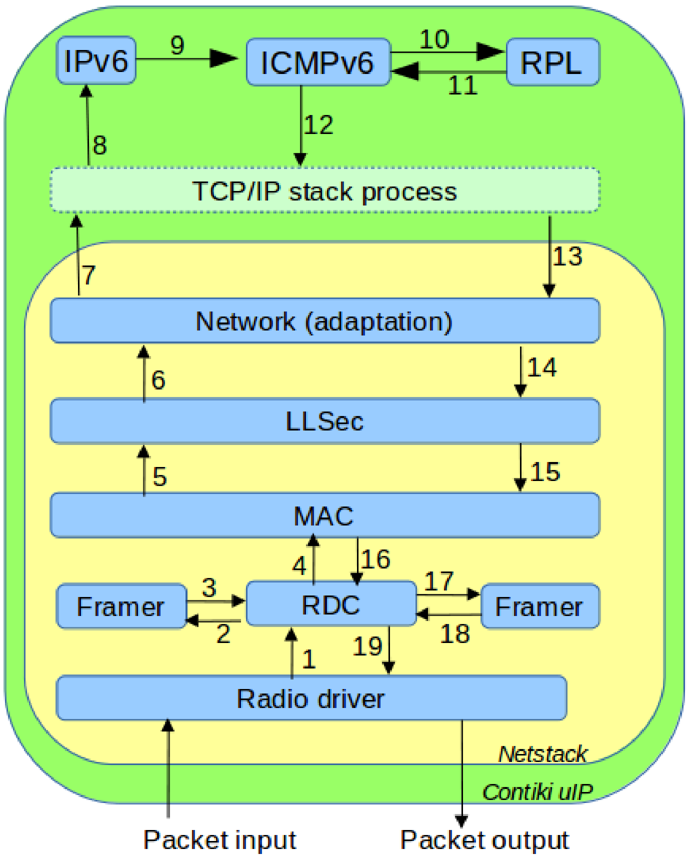


Figure 2.2: Netstack layers interaction

So, for each layer the semantic to be used is the following:

```
NETSTACK_LAYER.function( parameter )
```

RPL

The Contiki implementation of RPL supports two objective functions [37]: “Objective Function 0”, which provides a baseline for interoperability that only seeks to optimize the hop count routing metric, and the Minimum Rank Objective Function with Hysteresis (MRHOF), which exploits either the ETX metric or the energy metric. In particular, MRHOF uses hysteresis while selecting the path with the smallest metric value. The use of MRHOF with the Expected Transmission Count (ETX) metric allows RPL to find the stable minimum-ETX paths from the nodes to a root in the DAG instance. The MRHOF is designed to find the paths with the smallest path cost while preventing excessive churn in the network. It does so by using two mechanisms. First, it finds the minimum cost path, i.e., path with the minimum Rank, then it switches to that minimum Rank path only if it is shorter (in terms of path cost) than the current path by at least a given threshold. This second mechanism is called "hysteresis". More information on this algorithm can be found at [38].

2.2.2 Routing Tables

Contiki holds three main routing tables:

- Neighbor Cache: already presented in paragraph [ND6 Data Structures]. This structure holds the node’s neighbors
- Default Router List or Default Route List: already presented in paragraph []. In RPL, the default router for a given node corresponds to the neighbor which is closer to the border-router. In some cases, there could be more than one default router. In Contiki, a Default Router List entry is a default route
- Neighbor Routes: holds information about what routes go through what neighbor. It memorizes every route which is not the node’s default route, i.e. the packet forwarding rule to use when no specific route can be determined of a given destination address

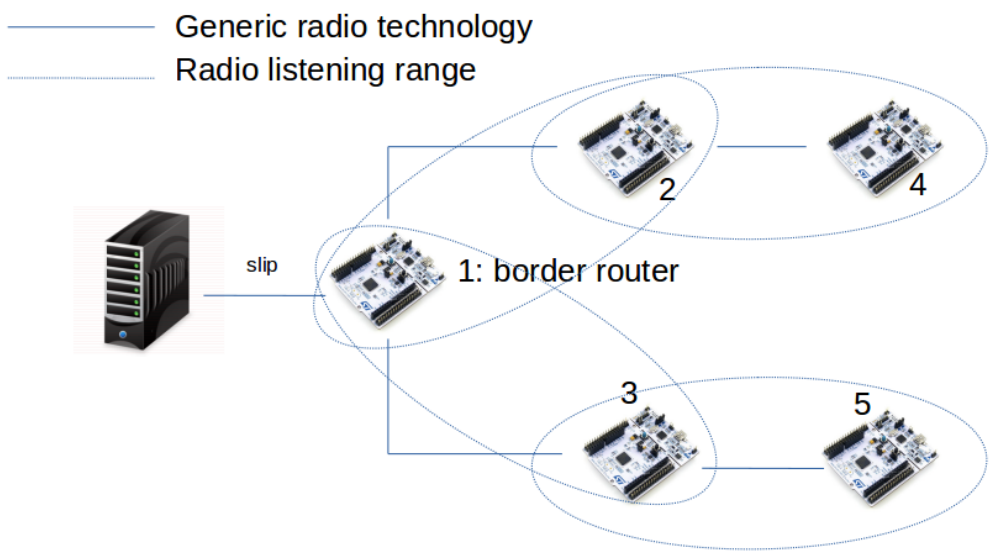


Figure 2.3: Network composition of the routing tables example

Below, it's illustrated an example of how these routing tables get filled on Contiki-based nodes in a real scenario. Figure 2.3 illustrates the network composition for the example.

The radio listening range for the nodes are represented by the dotted lines. Thus:

- border-router can listen to 2 and 3
- 2 can listen to 1 and 4
- 3 can listen to 1 and 5
- 4 can listen only to 2
- 5 can listen only to 3

Border-router's Neighbor Routes Table holds the following information:

- 2 is directly reachable, i.e. 2 is reachable via 2
- 3 is directly reachable, i.e. 3 is reachable via 3
- 4 is reachable via 2
- 5 is reachable via 3

No default route is held in the border-router Default Router List.

Border-router's Neighbor Cache holds the neighbors 2 and 3.

2's Neighbor Routes Table holds only the following information:

- 4 is directly reachable, i.e. 4 is reachable via 4

The default route held in 2's Default Router List is 1.

2's Neighbor Cache holds the neighbors 1 and 4.

4's Neighbor Routes Table is empty. 4's default route is 2. 4's Neighbor Cache holds only 2.

3 and 5 are specular to 2 and 4.

2.3 Other OSs

In this paragraph is presented other OSs that are relevant in the IoT.

ARMmbed

mbed is a platform and operating system for internet-connected devices based on 32-bit ARM Cortex-M microcontrollers, which devices are mainly known for their utilization in the IoT field. mbed is the industry's first online platform for fast, low-risk prototyping of microcontroller-based systems. mbed microcontroller was in 2010 the recipient of an EDN Innovation Award, which honors electronics engineers and the ground-breaking products they produce. The project is collaboratively developed by ARM and its technology partners.

The applications for mbed OS can only be developed online using its native online code editor cum compiler known as mbed online integrated development environments (IDEs); while writing of code can only be done through a web browser, its compilation is done by the ARMCC C/C++ compiler in the cloud.

mbed OS supports the following connectivity technologies: BLE, Wi-Fi, Zigbee, Ethernet and 6LoWPAN.

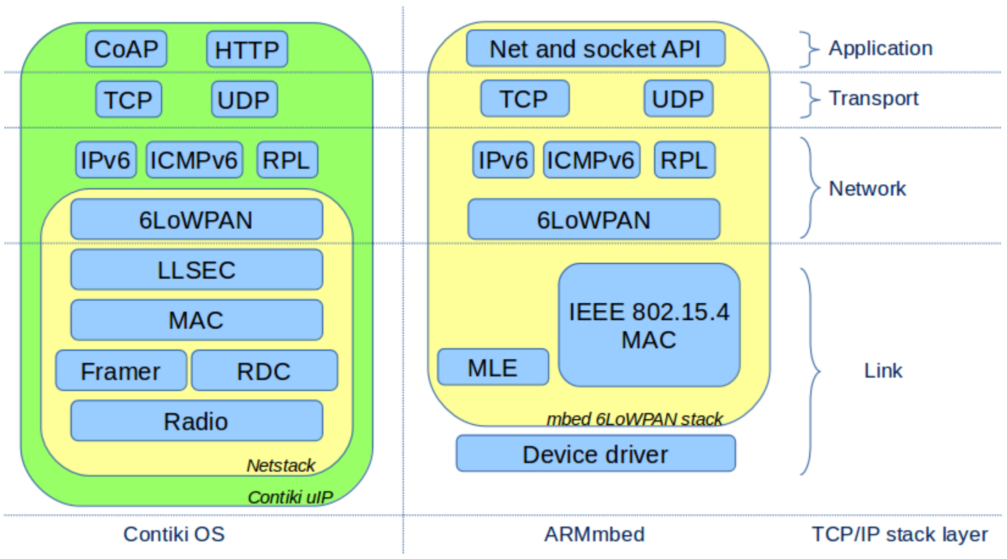


Figure 2.4: Contiki Netstack VS mbed network stack

The figure 2.4 puts side-by-side Contiki Netstack with ARMmbed Netstack in order to make a comparison between them.

As can be seen from the figure, they are very similar. Note that mbed IEEE 802.15.4 MAC could implement also LLSEC, if wanted.

The MLE module is present in mbed but not in Contiki. MLE stands for Mesh link Establishment and it operates below the routing layer and adds three capabilities to IEEE 802.15.4:

- Dynamically configuring and securing radio links. Assures anti-replay protection
- Enabling network-wide changes to radio parameters
- Detecting neighboring devices

The purpose of the third, detecting neighboring devices, is to make link management more efficient by detecting unreliable links before any effort is spent configuring them.

One of the main functions of MLE is to initialize link-layer security; this means that MLE itself cannot rely on link-layer security; to avoid the cost and complexity of adding a second security suite, MLE reuses that of 802.15.4.

All MLE messages are sent using UDP.

All the similarities between Contiki Netstack and mbed 6LoWPAN stack indicate a technology convergence, thus Contiki OS is a good choice for IoT projects.

TinyOS

Another alternative OS to Contiki is TinyOS: it's a free and open source software component-based operating system and platform targeting wireless sensor networks (WSNs). TinyOS is an embedded operating system written in the nesC programming language as a set of cooperating tasks and processes.

Main differences between TinyOS and Contiki are reported below [39, 40].

- Limited resources: both operating systems can be run on microcontrollers with very limited resources, but due to the higher complexity of the Contiki kernel TinyOS can generally get by with lower resource requirements
- Concurrency: TinyOS offers only the event-driven kernel as a way of fulfilling the concurrency requirements; while Contiki also uses an event-driven kernel it also has different libraries that offer different levels of multithreading on top of that
- Flexibility: whereas TinyOS is a monolithic system, Contiki is a modular system; in monolithic systems, an application is compiled with the OS as a monolithic program; on the other hand, in modular systems, it is compiled into an individual program module that is loadable by the OS kernel. Modular systems are more flexible when the individual application needs to be frequently modified through network reprogramming, for example when the node software has to be updated often for a large amount of nodes
- Low Power: TinyOS has out-of-the-box better energy conservation mechanisms but for Contiki similar power saving mechanisms can be implemented

Chapter 3

Proposed Solution

This chapter presents the main software changes made in Contiki OS and the main multi-interface network prototypes tested in this work. In the following the words “multi-interface” and “multiradio” are used equivalently. The “multiradio” denomination is due to the fact that Contiki calls the first level of its Netstack, that implements the physical driver, “Radio”. In this study the obtained working configurations exploit two communication technologies: the radio and the power-line. Thus, it is preferred the utilization of “multi-interface” instead of “multiradio”, even if both of them can be considered correct, from a Contiki terminology point of view.

Overview

Goal of this thesis is to create a 6LoWPAN-based network prototype that is formed by nodes with different communication interfaces.

By using Contiki development kit’s network applications examples, several working configurations have been tested, all composed by one dual-interface 6LoWPAN Border Router (6LBR), with a SPIRIT1 Radio and a ST7580 Powerline modem, and several mono-interface nodes, with alternatively a SPIRIT1 Radio or a ST7580 Powerline modem. The dual-interface 6LBR interacts with two subnets composed respectively by radio nodes and by powerline nodes. Hence, the 6LBR routes the information from one subnet to the other. Subnet prefixes are assigned as parameters via tunslip6 from the host to the 6LBR and then propagated through RPL mechanisms.

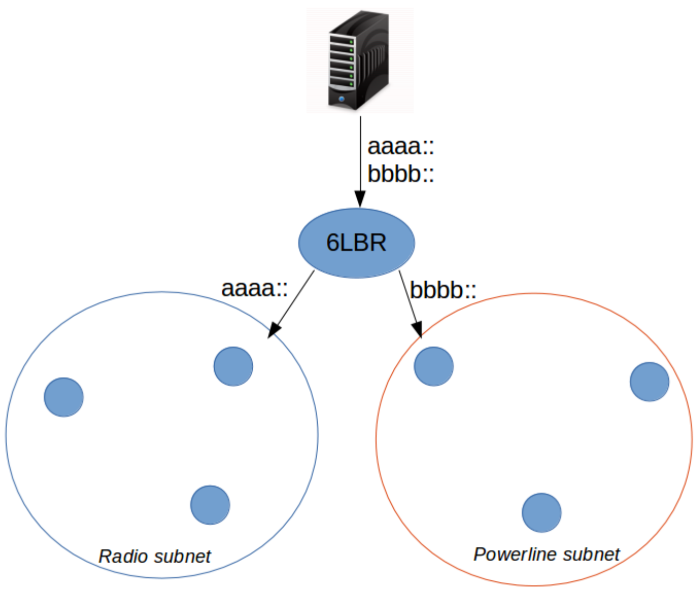


Figure 3.1: 6LBR dual-interface subnets

The main idea behind the new Contiki multi-interface feature is to vectorize every Contiki Netstack layer in order to allow multi-interface nodes to have a separated Netstack for every physical interface. In particular, all the Netstack layers arrays have a size equal to the number of the node's interfaces. Tunslip6 has been modified to allow more than one subnet prefix to be sent to the 6LBR in order to set interfaces' addresses and to create as many subnets as the number of the 6LBR's interfaces. ND6 and RPL have been adapted as well in order to correctly implement the multicast and unicast packet sending.

Next sections introduce the hardware implementation of the different working configurations, the main ideas behind the software changes and some concrete examples of source code modifications.

3.1 Hardware Implementation

This section presents the physical platforms chosen to achieve this work objectives and gives an overview of the most interesting working configurations useful to demonstrate the new Contiki multi-interface feature.

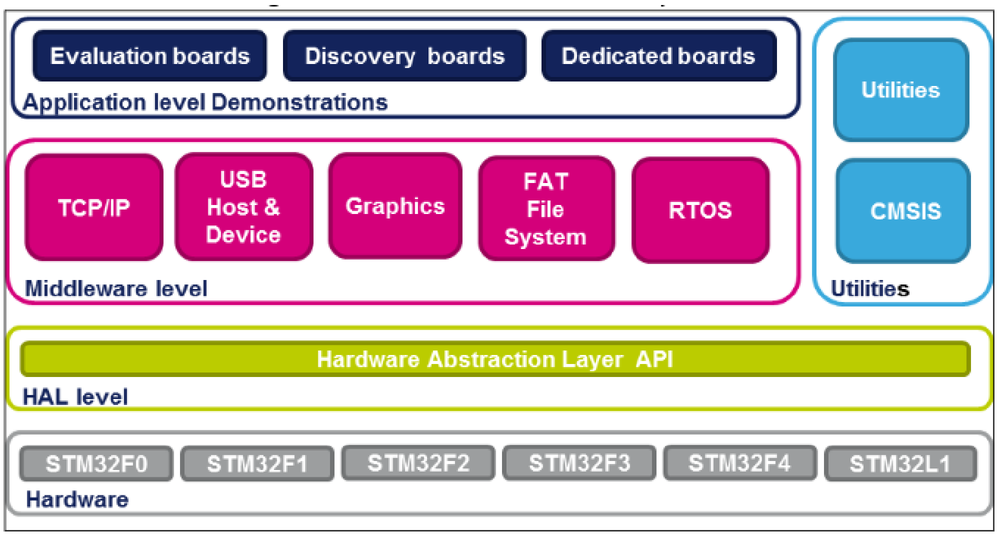


Figure 3.2: STM32 Cube L1 software package block diagram

STM32 Nucleo Development Platform

The STM32L152RE Nucleo programmable board has been chosen for this work. It is equipped with a STM32L152RE microcontroller by ST Microelectronics and is part of the STM32 Nucleo-64 family. This board provides Arduino connectivity support and some additional "Morpho" headers, allowing to expand the functionality of the base Nucleo board with a wide range of Arduino Expansion Board and STM32 Nucleo Expansion Boards that can be plugged on top of the STM32L152RE Nucleo PCB extending its capabilities.

The STM32L152RE microcontroller integrates an ARM Cortex M3 32bit RISC processor operating at a frequency of up to 32 MHz, a 512 KB Flash memory and a ST-LINK debugger/programmer interface.

The Nucleo board comes with the STM32CubeL1 software package, which includes all the software components required to develop an application on STM32L1 microcontrollers. The block diagram of STM32Cube is shown in figure 3.2:

The STM32Cube firmware solution is built around three independent levels that can easily interact with each other as shown in figure 3.3 on the next page.

Level 0 is divided in three sub-layers:

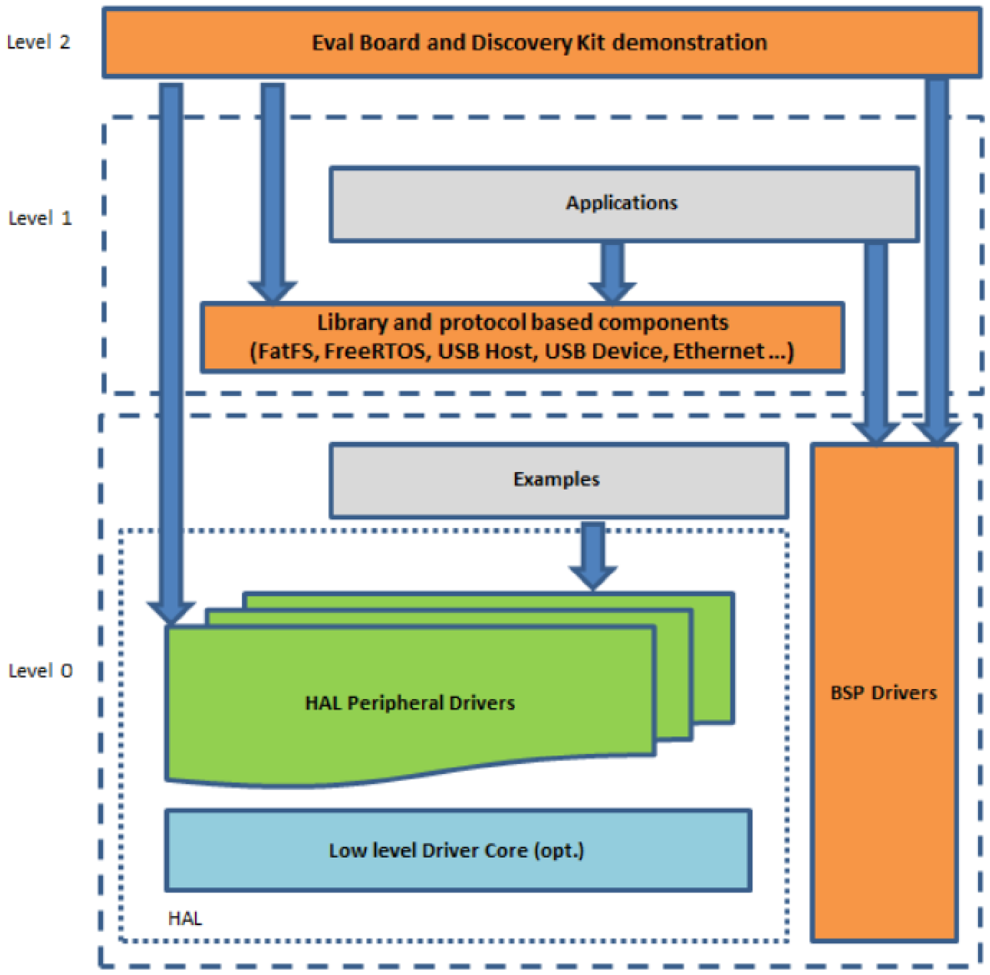

Level 2
Eval Board and Discovery Kit demonstration
Level 1
Applications
Library and protocol based components
(FatFS, FreeRTOS, USB Host, USB Device, Ethernet ...)
Level 0
Examples
HAL Peripheral Drivers
Low level Driver Core (opt.)
HAL
BSP Drivers

Figure 3.3: STM32 Cube L1 layers interaction

- Board Support Package (BSP): is based on modular architecture allowing an easy porting on any hardware by just implementing the low level routines. This layer offers a set of APIs relative to the hardware components in the hardware boards (LCD drivers, microSD, etc...), composed of two parts:
  - Component: is the driver relative to the external device on the board, not related to the STM32, the component driver provide specific APIs to the BSP driver external components, could be portable on any other board
  - BSP driver: it allows to link the component driver to a specific board, provides a set of friendly used APIs. The APIs naming rule is `BSP_FUNCT_Action()`, so there could be, for instance, `BSP_LED_Init()` and `BSP_LED_On()`
- Hardware Abstraction Layer (HAL): this layer provides the low level drivers, the hardware interfacing methods to interact with the upper layers (application, libraries, stacks). It provides generic, multi-instance, functionalities oriented APIs which permit to offload the user application implementation by providing ready to use process. As example, for the communication peripherals (I2S, UART...) it provides APIs allowing to initialize, configure the peripheral, manage data transfer based on polling, interrupt or DMA process, manage communication errors that may raise during communication. The HAL Drivers APIs are split in two categories, generic APIs which provides common, generic functions to all the STM32 series, extension APIs which provides specific, customized functions for a specific family or a specific part number
- Basic peripheral usage examples: this layer encloses the examples build over the STM32 peripheral using only the HAL, BSP resources

Level 1 is divided in two sub-layers:

- Middleware components: set of Libraries covering USB Device library, STMTouch touch sensing library, graphical STemWin library, FreeRTOS, FatFS. Horizontal interaction between the components of this layer is done directly by calling the feature APIs while the vertical interaction with the low level drivers is done through specific callbacks, static macros implemented in the library system call interface. As example, the FatFS implements the disk I/O driver to access microSD drive or the USB Mass Storage Class. The main features of each Middleware component are the following:

  - USB Device Library
    * Supports several USB classes (Mass-Storage, HID, CDC, DFU, AUDIO, MTP)
    * Supports multi packet transfer features: allows sending big amounts of data without splitting them into max packet size transfers
    * Uses configuration files to change the core, the library configuration without changing the library code (Read Only)
    * RTOS, Standalone operation
    * The link with low-level driver is done through an abstraction layer using the configuration file to avoid any dependency between the Library and the low-level drivers
  - FreeRTOS
    * Open source standard
    * CMSIS compatibility layer
    * Tickless operation during low-power mode
    * Integration with all STM32Cube Middleware modules
  - FAT File system
  - STM32 Touch Sensing Library
  - STemWin Library
    * Graphical library supporting LCD provided as part as the STM32CubeL1 firmware package

- Examples based on the Middleware components: each Middleware component comes with one or more examples (called also Applications) showing how to use it. Integration examples that use several Middleware components are provided as well

Level 2: is composed of a single layer which is a global real-time, graphical demonstration based on the Middleware service layer, the low level abstraction layer, the basic peripheral usage applications for board based functionalities.

STM32L152RE Nucleo Board is shown in picture 3.4 on the facing page.

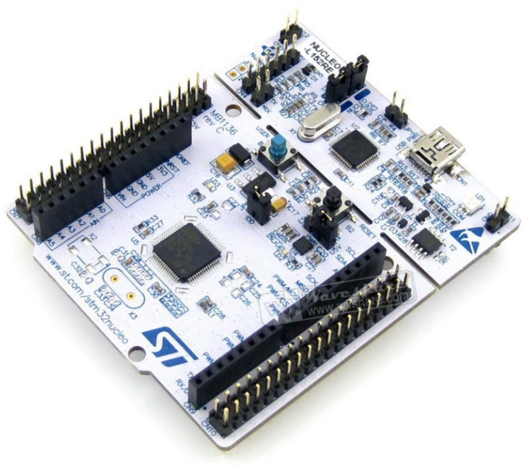


Figure 3.4: STM32 Nucleo L152-RE

The ultra-low-power STM32L152RE microcontroller operates from 1.8 to 3.6 V. The power supply to the entire board is provided either by the host PC through the USB cable, or by an external source.

The STM32L152RE microcontroller normally boot from system flash memory, executing a bootloader code and then the application firmware stored. The MCU user Flash memory can be flashed via one of its interfaces, which are: two I^{2}Cs, three SPIs, three USARTs, two UARTs and an USB.

More technical details about the STM32Nucleo L1 family are reported in figure 3.6 on the next page. Among them there are:

- a LCD driver
- two timers
- two watchdogs
- AES encryption
- several interfaces such as SPI, USB, USART and I^{2}C
- ADC / DAC devices

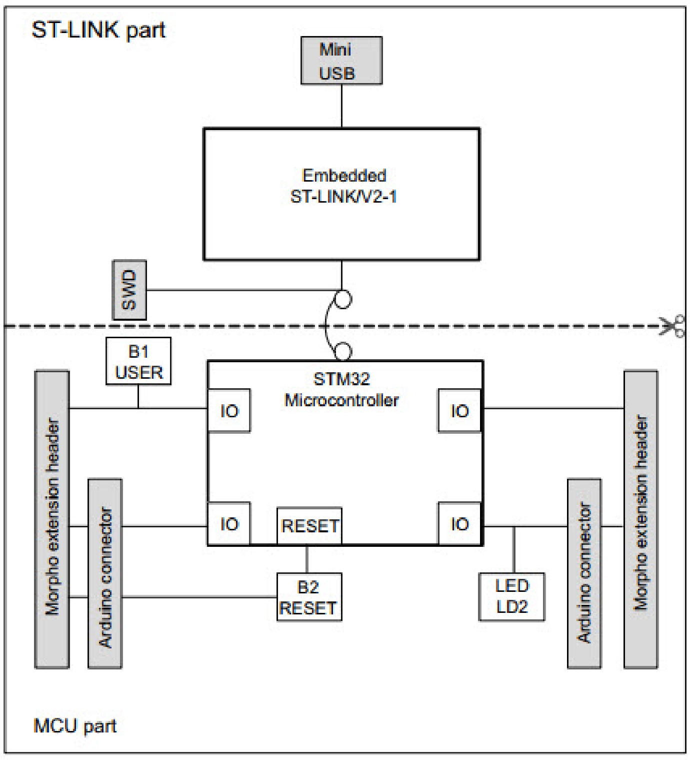


Figure 3.5: STM32L152RE main connectors

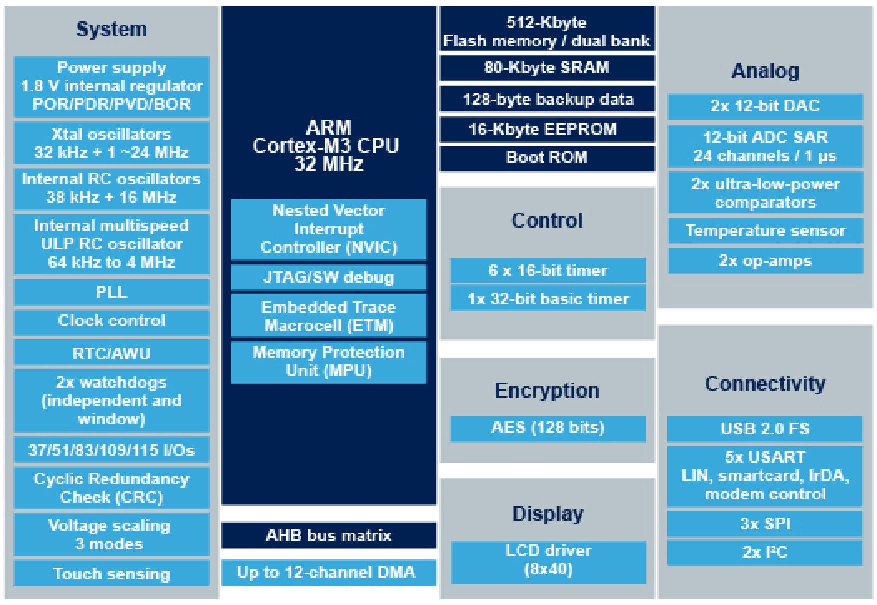

System
Power supply 1.8 V internal regulator POR/PDR/PVD/BOR
Xtal oscillators 32 kHz + 1 ~24 MHz
Internal RC oscillators 38 kHz + 16 MHz
Internal multispeed ULP RC oscillator 64 kHz to 4 MHz
PLL
Clock control
RTC/AWU
2x watchdogs (independent and window)
37/51/83/109/115 I/Os
Cyclic Redundancy Check (CRC)
Voltage scaling 3 modes
Touch sensing
ARM Cortex-M3 CPU 32 MHz
Nested Vector Interrupt Controller (NVIC)
JTAG/SW debug
Embedded Trace Macrocell (ETM)
Memory Protection Unit (MPU)
AHB bus matrix
Up to 12-channel DMA
512-Kbyte Flash memory / dual bank
80-Kbyte SRAM
128-byte backup data
16-Kbyte EEPROM
Boot ROM
Control
6 x 16-bit timer
1x 32-bit basic timer
Encryption
AES (128 bits)
Display
LCD driver (8x40)
Analog
2x 12-bit DAC
12-bit ADC SAR 24 channels / 1 µs
2x ultra-low-power comparators
Temperature sensor
2x op-amps
Connectivity
USB 2.0 FS
5x USART LIN, smartcard, IrDA, modem control
3x SPI
2x I²C

Figure 3.6: STM32L152RE technical details

SPIRIT1 Radio Expansion Board

The radio Expansion Board chosen to implement one of the two 6LBR physical interfaces is the IDS01A5 Nucleo Expansion Board, also named X-NUCLEO-IDS01A5, which is an evaluation board based on SPIRIT1 RF module SPSGRF-915. The SPSGRF-915 module operates in the 915 MHz ISM band. It is compatible with the ST Morpho and Arduino UNO R3 connector layout.

The SPIRIT1 is a very low-power RF transceiver, intended for RF wireless applications in the sub-1 GHz band. It is designed to operate in the license-free ISM frequency bands at 169, 315, 433, 868, and 915 MHz. Transmitted/received data bytes are buffered in two different three-level FIFOs (TX FIFO and RX FIFO), accessible via the SPI interface for host processing.

Communication with the MCU goes through a standard 4-wire SPI interface and 4 GPIOs. The device is able to provide a system clock signal to the MCU. MCU performs the following operations:

- Program the SPIRIT1 in different operating modes by sending commands
- Read and write buffered data, and status information from the SPI
- Get interrupt requests from the GPIO pins
- Apply external signals to the GPIO pins

The SPIRIT1 is configured by a 4-wire SPI-compatible interface (CSn, SCLK, MOSI, and MISO). More specifically:

- CSn: chip select, active low
- SCLK: bit clockMOSI: data from MCU to SPIRIT1 (SPIRIT1 is the slave)
- MISO: data from SPIRIT1 to MCU (MCU is the master)

As the MCU is the master, it always drives the CSn and SCLK. According to the active SCLK polarity and phase, the SPIRIT1 SPI can be classified as mode 1 (CPOL=0, CPHA=0), which means that the base value of SCLK is zero, data are read on the clock's rising edge and data are changed on the clock's falling edge. The MISO is in tri-state mode when CSn is high. All transfers are done most significant bit first.

The SPI can be used to perform the following operations:

- Write data (to registers or FIFO queue)
- Read data (from registers or FIFO queue)
- Write commands

The SPI communication is supported in all the active states, and also during the low power state: STANDBY and SLEEP. In order to notify the MCU of a certain number of events an interrupt signal is generated on a selectable GPIO.

ST7580 Power-line Expansion Board

The power-line Expansion Board chosen to implement the second 6LBR physical interface is the ST7580 Nucleo Expansion Board, also named `X_Nucleo_ST7580`. The ST7580 Expansion Board is not produced for the mass-market: it is an experimental prototype for testing purpose. The ST7580 device provides to the external host a complete physical layer (PHY) and some data link layer (DL) services for power line communication. It is mainly developed for smart metering applications in CENELEC A band, but suitable also for other control applications and remote load management in CENELEC B band.

A UART host interface is available for communication with an external host, exporting all the functions and services required to configure and control the device and its protocol stack.

Below is a list of the protocol layers and functions embedded in the ST7580:

- Physical (PHY) layer: hosted in the PHY processor, implements two different modulation schemes for communication through power line: a B-FSK modulation up to 9.6 kbps and a multi-mode PSK modulation with channel quality estimation, dual channel receiving mode and convolutional coding, delivering a throughput up to 28.8 kbps

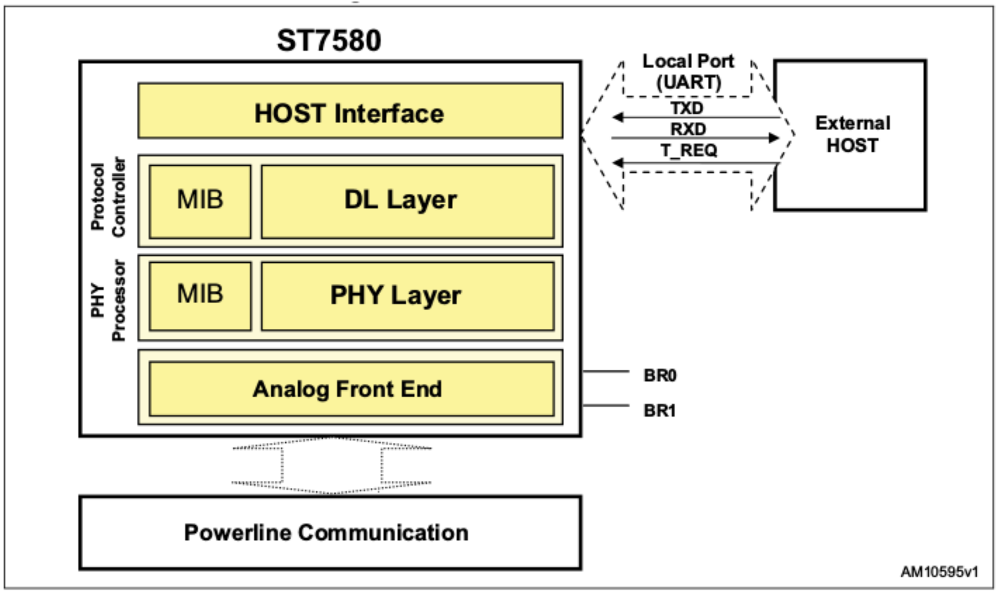


Figure 3.7: ST7580 powerline main protocols

- Data link (DL) layer: the embedded DL layer hosted in the protocol controller offers framing and error correction services. A further security service (SS) based on 128-bit AES algorithm is also available for crypting / decrypting frames

- Management information base (MIB): an information database with the data required for proper configuration of the system

- Host interface: all of the services of the PHY, DL and MIB are exported to an external host through the local UART port

IKS01A1 Sensors Expansion Board

This sensors Expansion Board is used only to implement one demo in order to have an interesting scenario for the obtained multi-interface 6LoWPAN prototype. The X-NUCLEO-IKS01A1 Expansion Board is a motion MEMS and environmental sensor evaluation board system. It is compatible with the Arduino UNO R3 connector layout, and is equipped with a 3-axis accelerometer, with a 3-axis gyroscope, with a 3-axis magnetometer and with a humidity, a temperature and a pressure sensor.

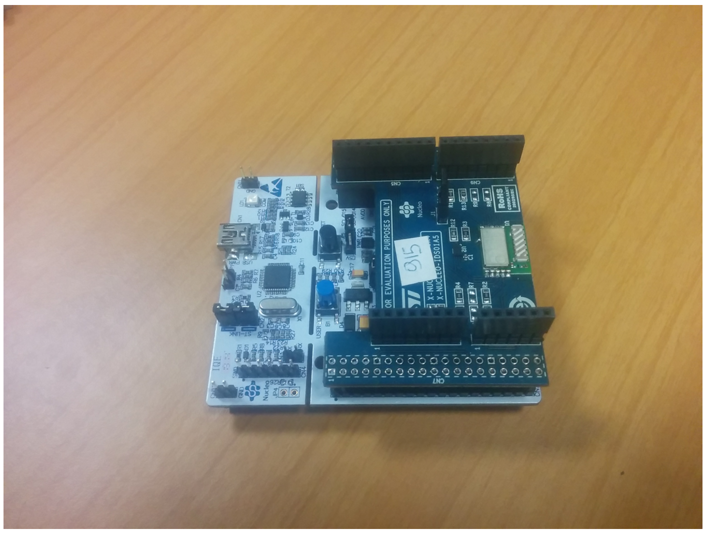


Figure 3.8: STM32NucleoL152RE with SPIRIT1

The X-NUCLEO-IKS01A1 interfaces with the STM32 microcontroller through an I^{2}C interface.

Hardware Working Configurations

As explained before, two different kind of nodes has been used to test the solution implemented:

- A dual-interface 6LoWPAN Border Router (6LBR), with a SPIRIT1 Radio and a ST7580 Powerline modem
- Mono-interface nodes, with alternatively a SPIRIT1 Radio or a ST7580 Powerline modem

The above nodes are physically composed by the following hardware combination:

- A STM32L152RE Nucleo with a SPIRIT1 Radio Expansion Board. The devices plugged together look like in the figure 3.8.
- A STM32L152RE Nucleo with a IDS01A5 SPIRIT1 Radio Expansion Board and with a IKS01A1 sensors Expansion Board. The three devices plugged together look like in figure 3.9 on the next page and 3.10 on the facing page

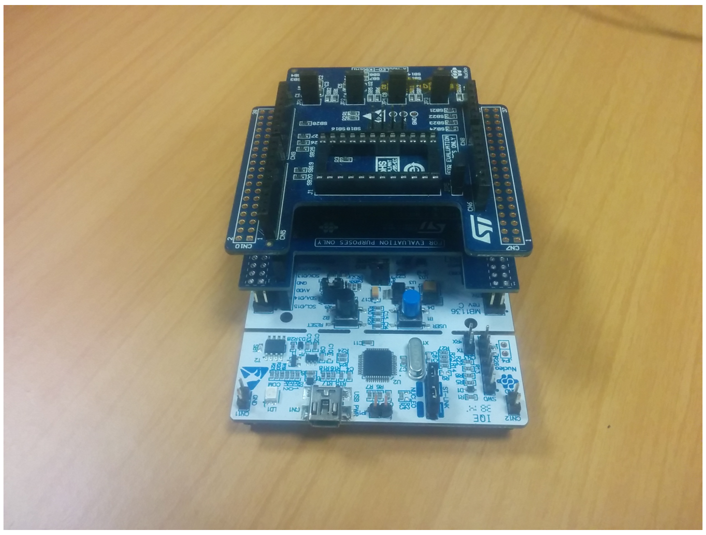


Figure 3.9: STM32NucleoL152RE with SPIRIT1 and sensors from front

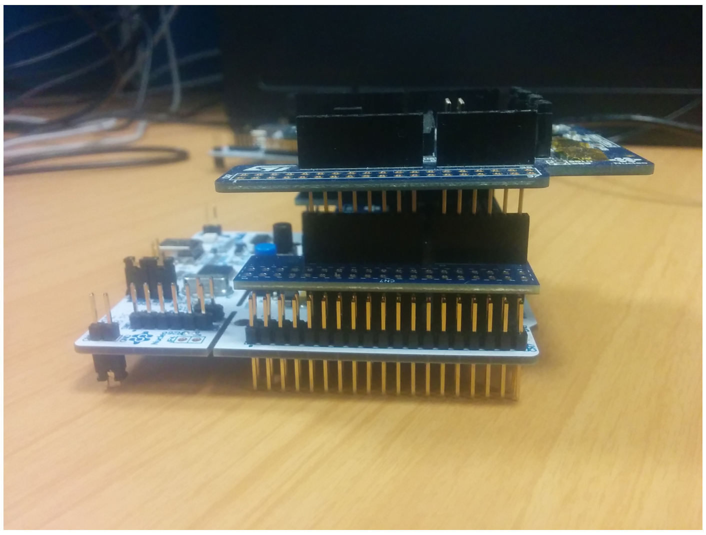


Figure 3.10: STM32NucleoL152RE with SPIRIT1 and sensors from side

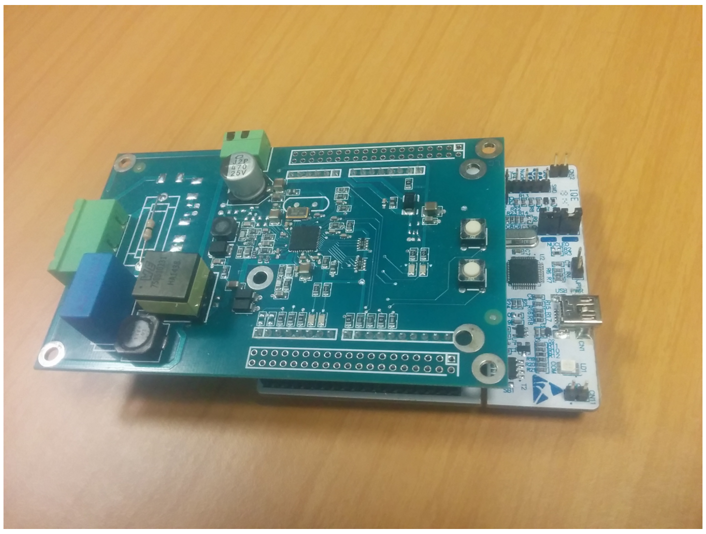


Figure 3.11: STM32NucleoL152RE with ST7580 powerline modem

- A STM32L152RE Nucleo with a ST7580 Powerline Expansion Board. The devices plugged together look like in figure 3.11.
- A STM32L152RE Nucleo with a IDS01A5 SPIRIT1 Expansion Board and with a ST7580 Powerline Expansion Board. The devices plugged and wired together look like in figure 3.12 on the facing page

In order to connect both the X-NUCLEO-IDS01A5 and the ST7580 for the 6LBR on the STM32L152RE Nucleo, some external wires that export the needed peripherals on the Arduino compatible connectors are needed. This is due to the lack of a sufficient number of GPIOs.

3.2 Software Modifications

This section describes the main software modifications applied to the Contiki vanilla code, to implement the "multiradio" solution.

3.2.1 Platform Porting

As stated in paragraph Tree Structure of chapter three, Contiki holds in the “platform” folder all the platforms that are supported by the OS,

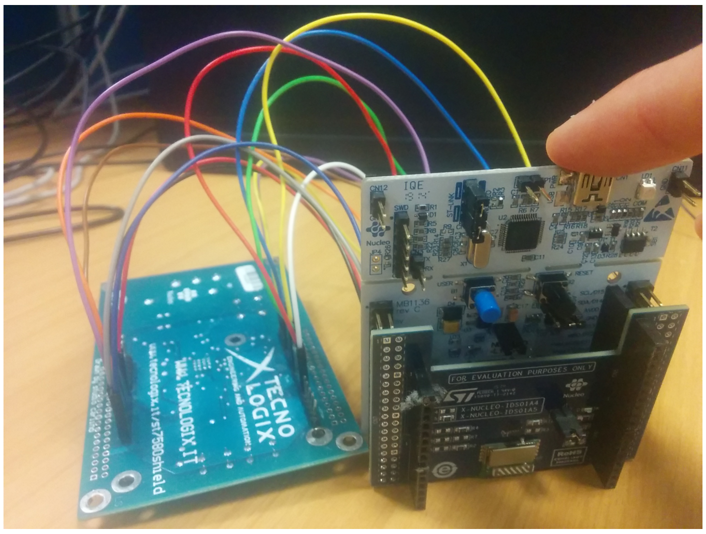


Figure 3.12: STM32NucleoL152RE with ST7580 powerline and SPIRIT1

including two ST platforms: the stm32nucleo-spirit1 and the stm32nucleo-st7580.

A new platform with the corresponding folder named "stm32nucleo-spirit1-st7580" has been added containing the porting code for a STM32L152 Nucleo provided with both the SPIRIT1 radio module and the ST7580 Powerline module, i.e. a multi-interface platform.

Makefile

The porting of Contiki on the new platform must follow this consideration: every time a driver of one of the two implemented module (SPIRIT1 and ST7580) updates, also the corresponding driver contained in the multi-interface platform folder must update. This is achievable by correctly setting up the Makefile to compile using directly the original code contained in the mono-interface platforms. The main sections of Makefile.stm32nucleo-spirit1-st7580 are reported in code listing 3.1.

Source Code 3.1: Makefile changes

```
#Retrieve the Spirit, x_nucleo_ids01ax, st7580 and x_nucleo_st7580 drivers in the relative platform folders;
#This way we want to exploit the already-existing and updated files and folders.
```

```
CONTIKI_TARGET_DIRS +=  ../stm32nucleo-spirit1/stm32cube-lib/drivers/spirit1/src \
    ../stm32nucleo-spirit1/stm32cube-lib/drivers/spirit1/inc
CONTIKI_TARGET_DIRS +=  ../stm32nucleo-spirit1/stm32cube-lib/drivers/x_nucleo_ids01ax
CONTIKI_TARGET_DIRS +=  ../stm32nucleo-st7580/stm32cube-lib/drivers/st7580/src \
    ../stm32nucleo-st7580/stm32cube-lib/drivers/st7580/inc
CONTIKI_TARGET_DIRS +=  ../stm32nucleo-st7580/stm32cube-lib/drivers/x_nucleo_st7580

[...]

# Duplicate headers issue: gcc will use the first headers it finds in the include paths, so the problem is solved by including the
# multi-interface platform before the monoradio one.
CFLAGS += -I. \
    -I$(CONTIKI)/cpu/arm/stm32l152 \
    -I$(CONTIKI)/core \
    -I$(CONTIKI)/platform/$(TARGET)/dev \
    -I$(CONTIKI)/platform/$(TARGET) \
    -I$(CONTIKI)/platform/$(TARGET)/stm32cube-lib/stm32cube-prj/Inc \
    -I$(CONTIKI)/platform/$(TARGET)/stm32cube-lib/drivers/Common \
    -I$(CONTIKI)/platform/$(TARGET)/stm32cube-lib/drivers/CMSIS \
    -I$(CONTIKI)/platform/$(TARGET)/stm32cube-lib/drivers/STM32L1xx_HAL_Driver/Inc \
    -I$(CONTIKI)/platform/stm32nucleo-spirit1/stm32cube-lib/drivers/spirit1/inc \
    -I$(CONTIKI)/platform/stm32nucleo-spirit1/stm32cube-lib/drivers/x_nucleo_ids01ax \
    -I$(CONTIKI)/platform/stm32nucleo-st7580/stm32cube-lib/drivers/st7580/inc \
    -I$(CONTIKI)/platform/stm32nucleo-st7580/stm32cube-lib/drivers/x_nucleo_st7580 \
```

Makefile changes allow the compiler to include during compilation the correct headers. In particular, the headers of the drivers must be retrieved in the mono-interface platform folders. Note that even the order of inclusion is important: the multi-interface headers must be included before the mono-interface ones, otherwise a "duplicate headers error" is thrown as gcc uses the first headers it finds in the include paths.

Main C Files

The new multi-interface platform folder contains the following C files:

- contiki-spirit1-st7580-main.c : this source file contains the main() function that is in charge of low-level drivers initialization (i.e. HAL drivers for the STM32Nucleo L152RE), clock initialization, main Contiki processes start and Netstack initialization

- contiki-conf.h : this file is the configuration file for Contiki. This file contains macros that enable or disable the different Contiki functionalities, that declare statically the drivers and protocols to be included by the Netstack, that configure some routing and radio parameters and many others

- hw-config.h : this file contains macros that mainly declare the GPIO utilization. The file has been approximately organized as follows:
  - USB GPIOs configuration
  - USART-UART GPIOs configuration
    - SLIP USART GPIOs configuration
    - ST7580 UART GPIOs configuration
  - SPIRIT1 SPI GPIOs configuration
  - I^{2}C GPIOs configuration: even if not used by the multi-interface 6LBR, I^{2}C GPIOs must be declared to avoid error signals by the STM32Cube-L1 library in compilation phase
  - Generic GPIOs configuration: SPIRIT1's and ST7580's digital input/output GPIOs (shutdown, reset, `T_REQ`, transmission and reception in progress)

- spirit1.c : in this file is defined the radio driver for SPIRIT1 used by the Netstack

- st7580.c : in this file is defined the "radio" driver for ST7580 used by the Netstack

Next paragraphs illustrate how these files are manipulated in order to obtain a functional multi-interface platform implementation.

3.2.2 RPL Subnetting

The main idea regarding the organization of a multi-interface 6LoWPAN is to have as many subnets as the number of the 6LBR's interfaces. The subnets prefixes are passed via tunslip6 through the SLIP connection from the host to the 6LBR. The 6LBR then distributes the prefixes by sending DIOs on the adiacent links. The nodes that are not directly connected to the 6LBR receive one of the prefixes distributed from the 6LBR via the DIO propagation made by the parent node. Note that it's not appropriate to use the terms association "link prefixes" to refer to the prefixes distributed by the 6LBR in the RPL net: in fact, the RPL links are 6LoWPAN links, i.e. links with single-hop reachability of neighboring nodes. All the nodes which exploit the same communication technology, for instance radio or powerline, are organized in the same subnet, i.e. are organized with the same prefix.

In the obtained working configurations, the 6LBR has two interfaces that correspond respectively to the SPIRIT1 radio link and the ST7580 power-line link. Thus, the 6LBR must distribute two prefixes: the aaaa:aaaa::1/64 prefix for the SPIRIT1 nodes and the bbbb:bbbb::1/64 prefix for the ST7580 Powerline nodes.

Tunslip6 Adaptations

A secondary idea regarding the organization of a multi-interface 6LoWPAN is to let the server hold a hard-coded IP address which is subnet-independent in order to let the client nodes communicate with him independently from its relative position (i.e. independently from the subnet he belongs to). This is not new in Contiki: the rpl-udp-client example implements a RPL client node which is aware of the server address because it is hard-coded, i.e. it's the aaaa::ff:fe00:1 address. The same address is hard-coded in the rpl-udp-server example. The main difference is that in the working configurations, the server holds two addresses: one auto-configurable address (via SLAAC) and one manually configured address which never changes its prefix. This implementation is named "IP aliasing" and is discussed in detail later.

Original tunslip6 usage is the following:

```
./tunslip [options] ipaddress
```

where an example is the following:

```
./tunslip6 -L -v2 -s ttyUSB1 aaaa::1/64
```

The prefix that tunslip6 takes as a parameter is the prefix that is assigned to all nodes of the RPL net. Obviously, the prefix is only one in the original Contiki implementation. The adapted tunslip6 main modifications takes two new options:

- **6LBR interfaces number option**: [-i numinterfaces] . This information allows in the command line the insertion of a number of prefixes equal to "numinterfaces" value. For instance, -i 2 allows a command line as the following:

  ```
  ./tunslip6 -i 2 -s ttyUSB1 aaaa:aaaa::1/64 aaaa:bbbb
      ::1/64
  ```

  where aaaa:aaaa::1/64 and aaaa:bbbb::1/64 are the two prefixes announced via -i 2

- **RPL server IP address**: [-r rplserveraddress] , where "rplserveraddress" is the server IP address to which clients send data. This option allows the host to ping the server IP address, i.e. add the server IP address to its routing table.

In Linux, the system kernel IPv6 routing table can be accessed via the following command line instruction:

```
route -6
```

which output the bash lines reported in table 3.1 on the following page when a RPL net is not running (i.e. the host current routing table).

The obtained working configurations has been tested with the following command line:

```
./tunslip6 -i 2 -s ttyUSB1 -r aaaa:abcd::ff:fe00:1 aaaa:
    aaaa::1/64 aaaa:bbbb::1/64
```

and the resulting host kernel routing table is reported in table 3.2 on the next page (RPL net up and running):

In this configuration, all the RPL nodes, i.e. even the server IP address can be pinged.

Table 3.1: Linux kernel IPv6 routing table without Contiki 6LoWPAN

Destination	Next-hop	Flag	Met	Ref	Use	If
fe80::/64	::	U	256	0	0	eth1
::/0	::	!n	-1	1	1	lo
::1/128	::	Un	0	3	65	lo
fe80::217:4ff:fe44:c01f/128	::	Un	0	1	0	lo
ff00::/8	::	U	256	0	0	eth1
::/0	::	!n	-1	1	1	lo

Table 3.2: Linux kernel IPv6 routing table with Contiki 6LoWPAN

Destination	Next-hop	Flag	Met	Ref	Use	If
aaaa::/64	::	U	256	0	0	tun0
aaaa:abcd::ff:fe00:1/128	::	U	256	0	0	tun0
bbbb::/64	::	U	256	0	0	tun0
fe80::/64	::	U	256	0	0	eth1
fe80::/64	::	U	256	0	0	tun0
::/0	::	!n	-1	1	1	lo
::1/128	::	Un	0	3	65	lo
aaaa::1/128	::	Un	0	1	0	lo
aaaa:abcd::ff:fe00:1/128	::	Un	0	1	0	lo
bbbb::1/128	::	Un	0	1	0	lo
fe80::/128	::	Un	0	1	0	lo
fe80::217:4ff:fe44:c01f/128	::	Un	0	1	0	lo
ff00::/8	::	U	256	0	0	eth1
ff00::/8	::	U	256	0	0	tun0
::/0	::	!n	-1	1	1	lo

DODAG Subnetting

Two main assumptions are behind the RPL net subnetting:

- A RPL instance is a net in which nodes can communicate among them. Thus, RPL subnets in which nodes can communicate among them must be part of the same RPL instance
- A RPL DAG root is a node which is the root for a DODAG. In ContikiRPL, a node can be the root only for one DODAG, i.e. a RPL instance must have different DAG root nodes. This assumption is implied by the "node's current DAG" implementation, that binds one node to a specific DAG in one temporal instant. When a new DODAG instance is detected, the node's current DAG changes. Thus, a node's current DAG is a DAG which belongs to a RPL instance. The RPL instance struct can be found in rpl.h header file.

Since one node belongs to just one DODAG in a temporal instant, RPL subnetting must happen inside a DODAG, i.e. within a node's current DAG, as the 6LBR must handle several subnets in the same temporal instant.

In order to better comprehend these considerations, Carel's virtual DODAG root implementation is illustrated below.

A **virtual DODAG root** [23] is the result of two or more RPL routers, for instance, 6LoWPAN Border Routers (6LBRs), coordinating to synchronize DODAG state and act in concert as if they are a single DODAG root (with multiple interfaces), with respect to the LLN. The coordination most likely occurs between powered devices over a reliable transit link. Carels's implementation [41] defines a new RPL "multi-sink" instance, i.e. many border routers that connect infrastructures over different locations/interfaces in one network. Every border router is logically binded to the other border routers via a parent virtual border router, and is physically connected to others border routers via wired connections to keep high performances. The resulting network is illustrated in figure 3.13 on the following page.

One main advantage of Carel's implementation is to have all the nodes organized in a single DODAG, which is easier to maintain with respect to different DODAGs since it requires no extra processing and no extra communication. Another main advantage is the memory usage: when

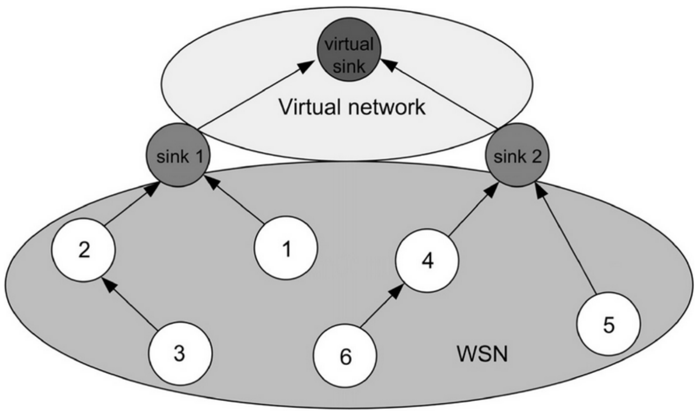


Figure 3.13: an example of a network with a virtual DODAG root

using a multiple DODAGs, each sink has to store the routes to each node in the network. When using a virtual sink, the memory usage to store the routes to children in the subtrees is spread over the different possible sinks.

From a certain point of view, the working configurations obtained from this work are a sort of real implementation of the virtual DODAG root on a single physical border router. The main differences are the following:

- there is not a virtual parent
- there is only one physical border router
- the border router has several interfaces, not just one
- the RPL net is organized in subnets, i.e. different prefixes are assigned to nodes which exploit different communication technologies

DODAG Prefix Field Vectorization

In order to handle the RPL subnetting, the main source code change was to vectorize the DAG prefix field. The new DAG structure with the multi-interface adaptation is illustrated in code listing 3.2.

`NETSTACK_CONF_NUMINTERFACES` represents the number of interfaces available on the node.

Source Code 3.2: RPL DAG

```
/* Directed Acyclic Graph */
```

```
struct rpl_dag {
  uip_ipaddr_t dag_id;
  rpl_rank_t min_rank; /* should be reset per DAG iteration
     ! */
  uint8_t version;
  uint8_t grounded;
  uint8_t preference;
  uint8_t used;
  /* live data for the DAG */
  uint8_t joined;
  rpl_parent_t *preferred_parent;
  rpl_rank_t rank;
  struct rpl_instance *instance;
#if NETSTACK_CONF_MULTIINTERFACES
  rpl_prefix_t prefix_info[NETSTACK_CONF_NUMINTERFACES];
#else
  rpl_prefix_t prefix_info;
#endif
  uint32_t lifetime;
};
```

In ContikiRPL one node belongs to one current DAG in every temporal instance. As can be seen from the `rpl_dag struct`, the DAG ID consists of one IP address: the IP address of the DAG root. In the tested working configurations the DAG root is always a multi-interface 6LBR, i.e. a node which has several IP addresses, one for each interface. The dilemma of which IP address to assign to the DAG ID is not a concrete problem: in fact, the DAG ID address is never used as a routing address, that's to say that any address could be used, with the only constraint that it must be unique inside a RPL instance. For this reason, the IP address relative to the 0-indexed interface can be used by default.

An RPL DAG holds a field named "prefix information" which specifies the prefix assigned to nodes that belong to the DAG. This field is filled via the DIOs exchange process, i.e. by copying the information held in the Prefix Information Option (PIO) of a DIO. Each time a DIO is received, if the prefix in the PIO is different from the prefix held in the DAG prefix info, the DAG prefix info takes the PIO value.

In order to implement the multi-interface feature on the 6LBR, the 6LBR's DAG must memorize all the prefixes that are active in the RPL net, organized by interface. Thus, the DAG prefix info field has been vectorized.

After the driver initialization, the border router sends to the host a prefix request via tunslip6. The host replies when the tunslip6 command line is executed. In the tunslip6 command line, the prefixes of the RPL subnets are passed as parameters, as discussed in Tunslip6 Adaptations section. These prefixes are then organized in an array, which is processed by the 6LBR as soon as the host reply arrives. These prefixes are memorized by the 6LBR in the DAG prefix info array field and are organized by interface: the prefix held in the 0-indexed cell of the array sent by the host is assigned to the 0-indexed DAG prefix info array cell, and so on. Then, the 6LBR sends as many DIOs as the number of its interfaces. Every DIO holds a PIO which corresponds to the relative interface: nodes reachable via the 0-indexed communication technology will receive a DIO with a PIO filled with the 0-indexed DAG prefix info array cell, and so on. Finally, the nodes utilize the PIO held by the DIOs to perform the SLAAC. Figures 3.14 on the facing page illustrate prefixes dissemination in a network composed by two subnets, composed in turn from only one mote.

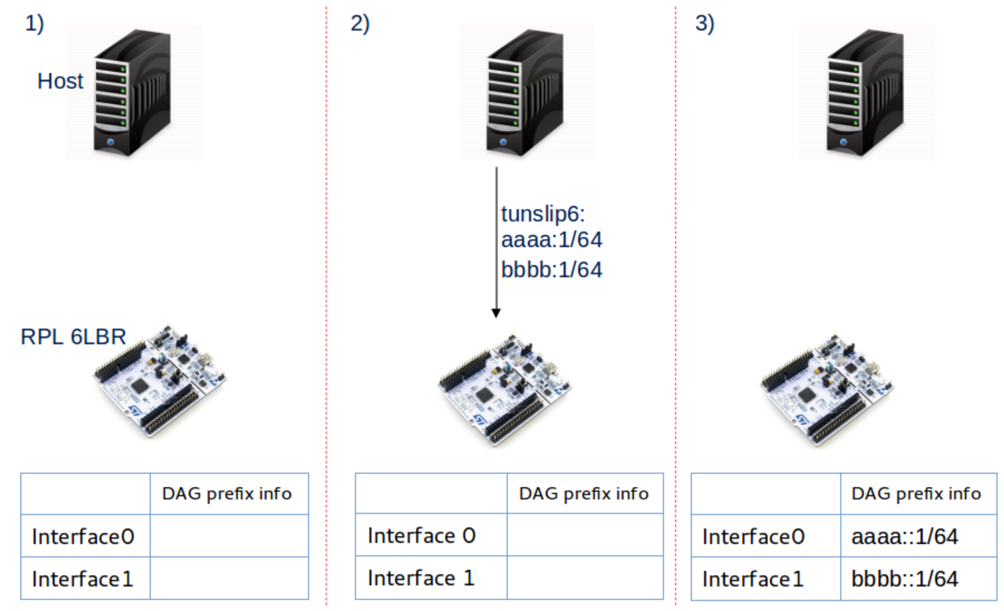
Even if the RPL doesn't utilize the "default" `uip_ds6_prefix_list`, which was originally conceived for the ND6 protocol, this prefix list is still used by a Contiki RPL-based application: in fact, this prefix list is filled during the data structure initialization (`uip_ds6_init()`) with the well-known link-local prefix, and it's consulted during packet integrity processing and during the next-hop determination phase. For this reason, it's logically coherent to vectorize the `uip_ds6_prefix_list` structure as illustrated in code listing 3.3.

Source Code 3.3: prefix list vectorization

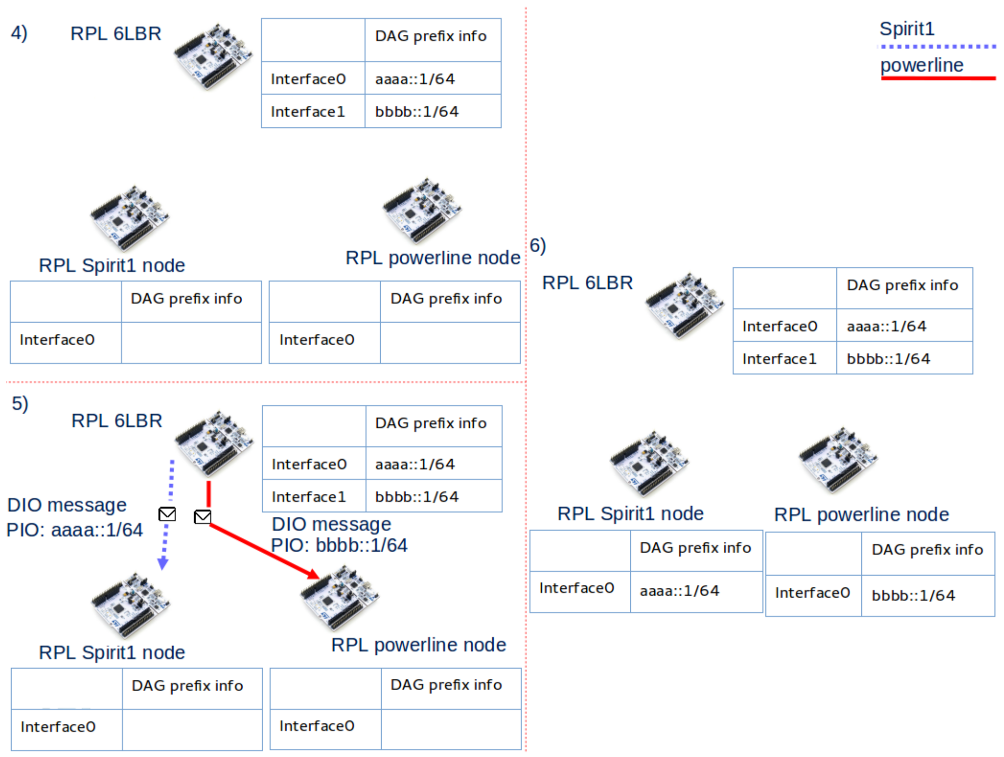
```
#if NETSTACK_CONF_MULTIINTERFACE
uip_ds6_prefix_t uip_ds6_prefix_list[
   NETSTACK_CONF_NUMINTERFACES][UIP_DS6_PREFIX_NB];
#else
uip_ds6_prefix_t uip_ds6_prefix_list[UIP_DS6_PREFIX_NB];
#endif
```

ND6 over 6LoWPAN Subnetting

ND6 over 6LoWPAN handles prefixes using the prefix list defined in `uip_ds6.h`. This prefix list has been already described in ND6 Data Structures section. As already said, the prefix list is filled from the RA prefix information. Thus, the principle is the same: the prefix list must be organized by interface, in order to retrieve the correct prefix information.



(a) *first phase.*



(b) *second phase.*

Figure 3.14: RPL prefixes dissemination in a network with two subnets

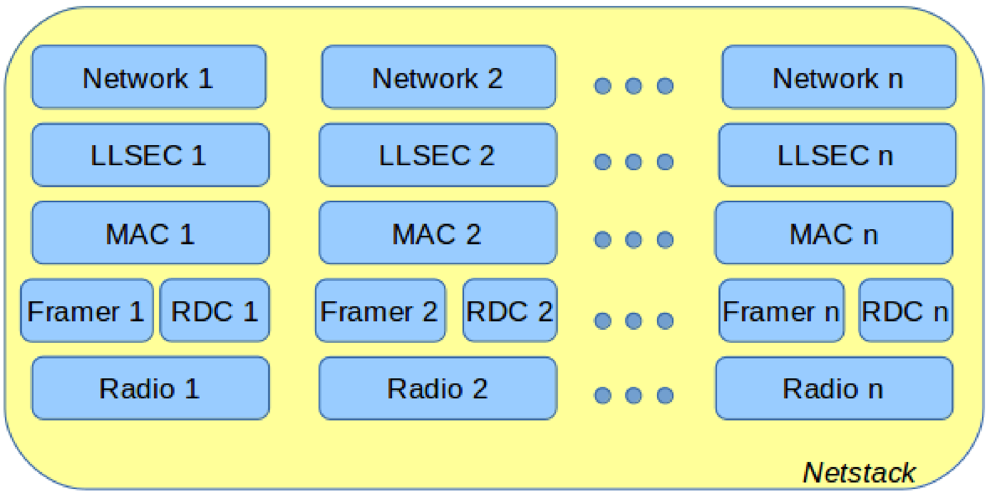


Figure 3.15: Netstack vectorized

The ND6 subnetting is not used in our testing configuration since all of them are RPL-based, i.e. the prefixes that are caught from the tunslip6 are processed by the RPL border router which sends the prefix array information to the DAG handler. Hence, in order to test the new multi-interface feature with ND6 a different network prototype should be conceived.

3.2.3 Netstack Vectorization

Contiki Netstack is a collection of drivers organized in a layered fashion. It's easy then to conceive a multi-interface Contiki version with a vectorized Netstack in which all the layers are arrays of drivers with a size equal to the number of the node's interfaces. Figure 3.15 illustrates this concept.

The choice of vectorize the network layer is justified by the fact that the Netstack is not exclusively used by uIPv6 stack. Even if 6LoWPAN could be a long-lived standard for IPv6 packet header compression, and even if can be assumed that no other adaptation layer protocols will be developed in the next years, the Netstack is used by other packet transmission mechanism. For instance, Rime network driver was developed in order to avoid utilizing IP protocol when resources are even more constrained. Thus, there could be other interesting network drivers to be implemented by Contiki in the near future. Same goes for the MAC layer: the IEEE 802.15.4 states that the standard MAC driver for LR-WPAN should be

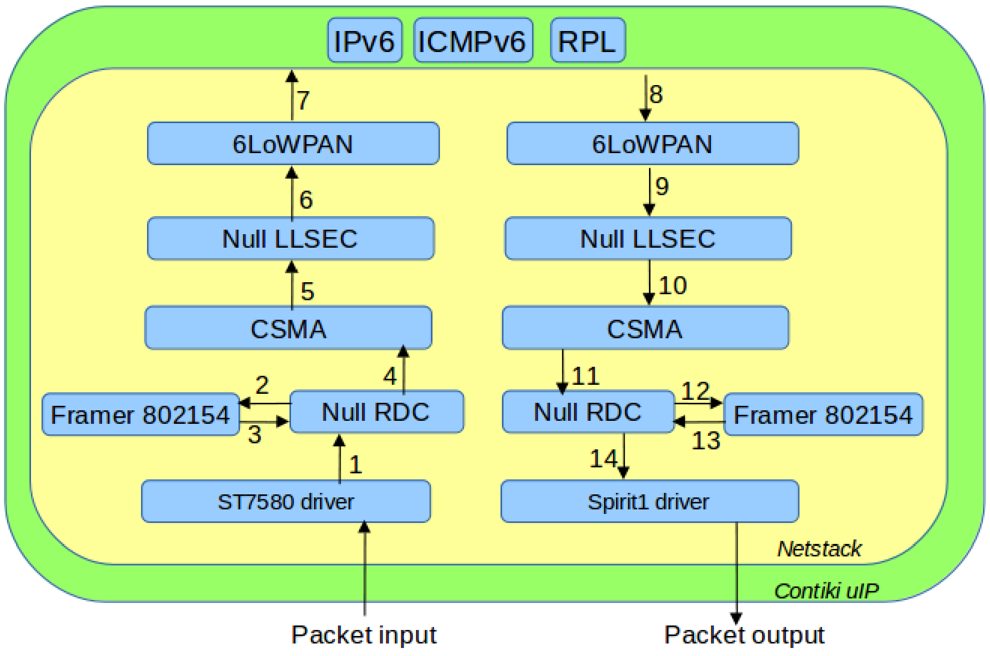


Figure 3.16: new layers interaction during a "receive-and-reply to a packet" event

the CSMA/CA protocol, but again, in some cases is useful to adopt some ad-hoc protocols in combination (Null MAC driver + ContikiMAC RDC driver) in order to fulfill the constraints or to achieve the objectives.

The new layer interaction scheme in the most frequent system event, i.e. the "receive-and-reply to a packet" event, is illustrated in figure 3.16.

Contiki Netstack protocols illustrated in figure 3.16 are the protocols implemented by the nodes in the working configurations obtained in this work. Hence, the protocols are the same for each node with the exception of the radio driver which is different depending on the physical interface of the node. Thus, the SPIRIT1 nodes exploit the SPIRIT1 radio driver, the ST7580 nodes exploit the ST7580 radio driver and the 6LBR node exploits both the drivers. Even if in the obtained working configurations the 6LBR Netstack protocols are the same with the exception for the interface driver, Netstack drivers could be different for each 6LBR interface. The idea is exactly to have different interfaces which implement different Netstack protocols. Finally, Netstack arrays are implemented as illustrated in code listing 3.4 on the next page.

Source Code 3.4: new Netstack arrays

```c
struct llsec_driver   NETSTACK_LLSEC[NETSTACK_CONF_NUMINTERFACES];
struct mac_driver NETSTACK_MAC[NETSTACK_CONF_NUMINTERFACES];
struct rdc_driver   NETSTACK_RDC[NETSTACK_CONF_NUMINTERFACES];
struct framer   NETSTACK_FRAMER[NETSTACK_CONF_NUMINTERFACES];
struct radio_driver   NETSTACK_RADIO[NETSTACK_CONF_NUMINTERFACES];
```

where NETSTACK_CONF_NUMINTERFACES is a macro that contains the number of the links above which the 6LBR interfaces himself, i.e. the number of the node's interfaces. This declaration goes in the netstack.c file, which contains the details of the Netstack. The array initialization for a dual-interface node is illustrated in code listing 3.5.

Source Code 3.5: dual-interface node Netstack arrays initialization

```c
  /* SPIRIT1 Netstack arrays initialization. */
  NETSTACK_LLSEC[SPIRIT1_INTERFACEID]  = nullsec_driver;
  NETSTACK_MAC[SPIRIT1_INTERFACEID]    = csma_driver;
  NETSTACK_RDC[SPIRIT1_INTERFACEID]    = nullrdc_driver;
  NETSTACK_FRAMER[SPIRIT1_INTERFACEID] = framer_802154;
  NETSTACK_RADIO[SPIRIT1_INTERFACEID]  = spirit_radio_driver;

  /* ST7580 Netstack arrays initialization. */
  NETSTACK_LLSEC[ST7580_INTERFACEID]  = nullsec_driver;
  NETSTACK_MAC[ST7580_INTERFACEID]    = csma_driver;
  NETSTACK_RDC[ST7580_INTERFACEID]    = nullrdc_driver;
  NETSTACK_FRAMER[ST7580_INTERFACEID] = framer_802154;
  NETSTACK_RADIO[ST7580_INTERFACEID]  = st7580_radio_driver;
```

where SPIRIT1_INTERFACEID and ST7580_INTERFACEID are macros which contains respectively the values of 0 and 1. This initialization can be found in the main file relative to the new multi-interface platform.

As already explained, Contiki offers a high level of system customization. In fact, the source code is built with customizable macros and code blocks inserted into "if" conditional directives. At compile-time, the code blocks which are inside a false condition of a "if" directive are not compiled, i.e. not ported on the binary of the application. Hence, the idea was to follow this programming model to implement the new Contiki multi-interface feature: this software improvement can either be implemented or not simply by setting a macro value to 1. This macro has been named NETSTACK_CONF_MULTIINTERFACE. All the new code blocks are inserted inside the "if" directive as illustrated in 3.6.

Source Code 3.6: new Contiki source code organization

```
#if NETSTACK_CONF_MULTIINTERFACE

  //New multi-interface feature code

#else

  //Original Contiki code

#endif
```

The NETSTACK_CONF_MULTIINTERFACE and NETSTACK_CONF_NUMINTERFACES macros are defined in the contiki-conf.h header file. In particular, in the multi-interface 6LBR the macros are defined as illustrated in code listing 3.7.

Source Code 3.7: new multi-interface feature's main macros

```
#define NETSTACK_CONF_MULTIINTERFACE 1
#define NETSTACK_CONF_NUMINTERFACES 2 //SPIRIT1 and ST7580
    Powerline
```

Neighbor-Interface Association

In order to implement the multi-interface feature, new information must be added to the Contiki data structures. This new information is basically: “who is reachable via a certain link” and “who can reach me via a certain link”. These information, from a logic point of view, can be synthesized in one single information: “which links (technology) are exploited by my neighbors”. In fact, a node is only interested to know how to reach the next-hop during a packet transmission. It has no interest in knowing which communication technology a node far from him utilizes, it is enough for him to know that a certain far node exists and that is reachable via one of its neighbors.

Actually, this “which links are exploited by my neighbors” information is so important for a node that it is not limited only to the packet transmission phase: this information affects all the packet input and output phase, i.e.

the Netstack upward and downward crossing phases. As figure 3.16 on page 125 shows, when a packet arrives from a link, the physical layer must process and forward the packet to a certain RDC driver, i.e. the RDC driver which is relative to that physical interface. It could be for instance the Null RDC driver for a generic radio A or could be the ContikiMAC RDC driver for a generic radio B.

Hence, the information “which link is exploited by a neighbor”, i.e. the interface information, must be related to the data structure that holds the neighbor main attributes. This data structure can be found in the kernel inside the network “net” folder, which contains all the network protocols and data structures. From there, every source file, protocol or data structure that is mentioned it’s assumed to be handled in some file inside the “net” folder of the kernel. The specific files which contain the data structures related to the neighbors are the uip-ds6-nbr.h and uip-ds6-nbr.c files. The main data structure in these files is the “neighbor cache”, which was already presented in the ND6 Data Structure section. The `uip_ds6_nbr_t` is the structure which describes a neighbor characteristics. Here is added the interface information in order to fulfill the neighbor-interface association.

The Neighbor Cache, the Default Router List and the Neighbor Routes Table are not vectorized, i.e. there is not the need to organize them by interface in order to implement the multi-interface feature. The reason is that during the routing, a node must be aware of all its neighbors and of all the routes that go through its neighbors. Thus, the logic behind the choice of the next-hop during a packet output stays the same.

Interface ID

The interface information has been simply translated in an integer value that uniquely identifies some interface and that can be named “interface ID”. This interface information is used mainly during the packet output and input phases, when some functions of some Netstack protocol are invoked, as illustrated in code listing 3.8.

Source Code 3.8: new Netstack layers’ functions invocation

```
NETSTACK_LAYER[interface_id].some_function()
```

The interface ID integer can take values from 0 (as the first array cell has index 0) to `NETSTACK_CONF_NUMINTERFACES`. Assuming that the number of interfaces can never be realistically more than a dozen, interface

ID can be declared as a `uint8_t` variable, i.e. the ARM small integer optimized version (unsigned and with 8 bit of representation, i.e. up to 255).

In order to uniquely identify an interface on a node, this information must be statically defined in the contiki-conf.h file. The macros in code listing 3.9 do the job.

Source Code 3.9: interfaces identifiers declaration

```
#define SPIRIT1_INTERFACEID 0
#define ST7580_INTERFACEID 1
```

These macros are used in the respective "radio" driver files (spirit1.c and st7580.c) in order to call the RDC driver:

```
NETSTACK_RDC[SPIRIT1_INTERFACEID].input();
```

in the spirit1.c source file, and

```
NETSTACK_RDC[ST7580_INTERFACEID].input();
```

in the st7580.c source file.

Interface ID as a Packetbuf Attribute

As illustrated in Neighbor-Interface Association section, the interface information, i.e. the interface ID, is saved in the neighbor cache. In order to do this, the interface ID, which is extrapolated from the physical layer during packet input, must be preserved until a new neighbor is added into it. This happens exactly after a packet input event: in fact, a new neighbor is detected when a packet is sent to the node and, after processing the packet, it comes out that the sender is not in the neighbor cache. In particular, the `uip_ds6_nbr_add(...)` function is called at the network layer, above the adaptation layer, i.e. out of the Netstack. This means that the interface information must cross the Netstack from the physical layer (i.e. from the radio driver) upward to the network layer (adaptation layer), then must reach the RPL and ND6 routing protocols, where the `uip_ds6_nbr_add(...)` is invoked.

The packetbuf global buffer saves a new incoming packet. This buffer is conceived to preserve the packet information while crossing the Netstack in order to let all the layers process the new packet. Furthermore, the

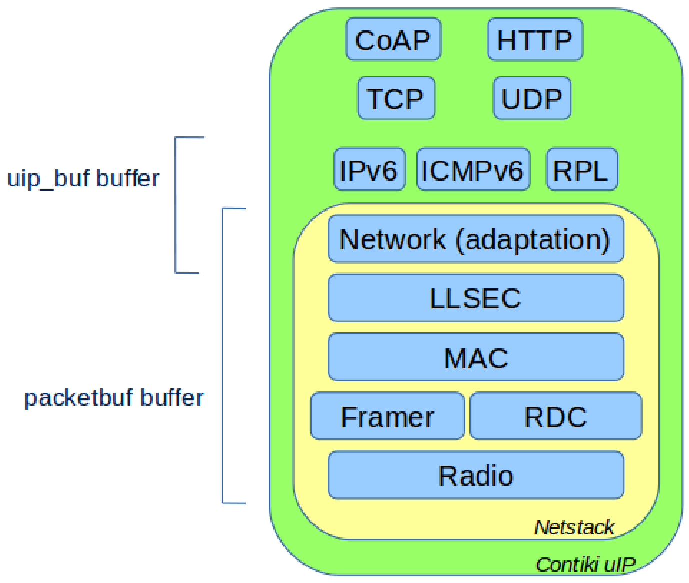


Figure 3.17: Contiki uIP's main global buffers

packet's information is organized in packet attributes called "packetbuf attributes". These attributes are distinguished in:

- scope 0 attributes, which are used only on the local node
- scope 1 attributes, which are used between two neighbors only
- scope 2 attributes, which are used between end-to-end nodes

In the figure 3.17 it's illustrated the scope of the two main Contiki global buffers.

Thus, a new packetbuf attribute, named `PACKETBUF_ATTR_INTERFACEID`, has been added to the scope 0 attributes. This attribute is set in the radio driver during packet input processing, with the function illustrated in code listing 3.10.

Source Code 3.10: interfaceID packetbuf attribute assignment to incoming packet

```
packetbuf_set_attr(PACKETBUF_ATTR_INTERFACEID,
    SPIRIT1_INTERFACEID);
```

During a packet output, the interface ID packetbuf attribute is set in the network driver, in order to allow the downward crossing of the Netstack.

This way, across the Netstack drivers the upward-downward layer invocation looks like the code listing 3.11:

Source Code 3.11: interfaceID packetbuf attribute usage

```
NETSTACK_LAYER[packetbuf_attr(PACKETBUF_ATTR_INTERFACEID)].
    function();
```

where `packetbuf_attr(interface_id)` is the “get attribute” function.

In order to let the interface ID reach the routing protocols, a new global variable is declared as follows:

```
uint8_t uip_ds6_interfaceid
```

This variable is used during packet input by the network driver in order to let the interface ID reach the routing protocols, and during packet output by the routing protocols in order to let the interface ID reach the network adaptation layer.

3.2.4 interface Structure Vectorization

Next phase was to vectorize the structure which holds an interface attributes. Such structure can be found in the uip-ds6.c source file, which holds all the main IPv6 related data structures. The vectorization it's illustrated in code listing 3.12.

Source Code 3.12: Interface structure vectorization

```
uip_ds6_netif_t uip_ds6_if[NETSTACK_CONF_NUMINTERFACES];
```

For every instance of `uip_ds6_if`, code must be organized with the "if" directive as shown previously. The interface structure is used in a lot of circumstances, thus only the main conceptual changes are reported. In order to exploit the desired interface, a lot of functions need to receive the interface ID as a parameter, so in general the main adaptation that has been applied to functions is illustrated in code listing 3.13.

Source Code 3.13: general functions adaptation

```
return_value_type
#if NETSTACK_CONF_MULTIINTERFACE
function(.., uint8_t interface_id)
#else
function(..)
```

```
#endif
{ function_body }
```

Interfaces Address Lists

Every interface holds an address list, as explained in IPv6 Node's Required Addresses section. Thus, every time that the system adds an address, it must add it to the correct interface. As a link-local address is formed by combining a well-known prefix (fe80::) with an interface identifier, the interfaces' address lists hold the same link-local address. This is allowed by the fact that an interface identifier may be used on multiple interfaces on a single node, as long as they are attached to different subnets.

Together with the link-local address, every interface's address list holds a global address, which is at first saved during the RPL examples initialization (i.e. the rpl-udp examples -server and client- and the rpl-border-router example) with the aaaa:: default prefix, and then eventually replaced during the Stateless Address Autoconfiguration process (SLAAC), i.e. when the 6LBR multicast to the nodes the prefix addresses received from the SLIP interface via tunslip6 (the 6LBR must eventually change its global addresses as well).

The function:

```
uip_ds6_addr_t *uip_ds6_addr_lookup(uip_ipaddr_t *ipaddr)
```

retrieves an address structure from the address list by seeking the IP address. This function is invoked in some cases for detect if a packet's target address is one of the node's addresses. That's the case for instance during a NS input processing: if the target address is one of the node's addresses, the node must respond to the NS with a NA packet. There could be cases in which the packet's target address is in a list that corresponds to an interface different from the one from which the packet entered. For example, if a packet sent on the ST7580 Powerline has `ADDRESS_X` as target address, it's not correct to look for `ADDRESS_X` in the ST7580 Powerline interface's address list, instead it's correct to seek in every interfaces' address lists. Hence, the search cycles on all the node's interfaces' lists and not only on one interface's list. For this reason, this function does not accept a interface ID parameter, but instead cycles on all the lists and then returns the address structure in which could be found the interface ID information contained in a new field, as illustrated in figure 3.14.

Source Code 3.14: new unicast address structure with interface ID field

```c
typedef struct uip_ds6_addr {
  uint8_t isused;
  uip_ipaddr_t ipaddr;
  uint8_t state;
  uint8_t type;
  uint8_t isinfinite;
  struct stimer vlifetime;
#if UIP_ND6_DEF_MAXDADNS > 0
  struct timer dadtimer;
  uint8_t dadnscount;
#endif /* UIP_ND6_DEF_MAXDADNS > 0 */
#if NETSTACK_CONF_MULTIINTERFACE
  uint8_t interface_id;
#endif
} uip_ds6_addr_t;
```

The multicast and anycast addresses are held in separated lists, which are named respectively multicast address list and anycast address list.

Multicast address list holds:

- one well-known all-nodes multicast address (ff02::1); every interface holds the same well-known all-nodes multicast address
- one well-known all-routers multicast address (ff02::2), if the node is a router; every interface holds the same well-known all-routers multicast address
- as many solicited-node multicast addresses as the number of its unicast addresses, which are obtained as described in Pre-defined Multicast Addresses section
- one well-known all RPL nodes multicast address (ff02::1a)

Anycast list in Contiki 3.0 is not used, i.e. there's no occurrence of the `uip_ds6_aaddr_add(..)` function anywhere in the source code.

Source Address Selection

The original sas function is defined in uip-ds6.c source file and is the following:

```
void uip_ds6_select_src(uip_ipaddr_t *src, uip_ipaddr_t *
   dst)
```

The original sas function takes two parameters: the source IP address field reference and the destination IP address field reference of the uIP global buffer. The source IP address field is the one which is filled by the sas algorithm (that's the reason why this parameter is passed by reference, as it is directly manipulated by the function), the destination IP address field is already defined and is used by the sas algorithm to choice the correct source address.

As specified in Source Address Selection section, routing (more precisely, selecting an outgoing interface on a node with multiple interfaces) is done before source address selection (sas). Since Contiki was conceived as a mono-interface system, this assumption was not followed at all. In fact, the sas in the original Contiki is done before the next-hop determination, i.e. before that an outgoing interface is selected. The next-hop determination is found in the `tcpip_ipv6_output()` function in the tcpip.c file, in the "ip" folder.

This issue must be solved differentiating two main cases:

- multicast output: a packet must be sent via all the interfaces that the 6LBR has. This is logically identifiable with the code implementation illustrated in code listing 3.15.

  Source Code 3.15: new multicast output implementation

  ```
  for(interface_id=0; interface_id <
     NETSTACK_CONF_NUMINTERFACES; interface_id++)
    packet_output(interface_id);
  ```

  i.e. the packet output function (for instance, the RS output and the DIO output) must be called as many times as the number of interfaces. In particular, the interface ID, which corresponds to the for cycle incrementing variable, must be passed to the function in order to let the uIP stack protocols correctly work. Multicast output concrete cases are discussed later

- unicast output: a packet must be sent to a specific address. This requires a way to retrieve the next-hop's interface ID in order to let the packet reach its final destination. The interface ID information in particular must be given in input to the sas in order to apply the algorithm on the right interface

In order to solve the unicast output issue, it has been implemented a new function that does the next-hop determination as the `tcpip_ipv6_output()` function does and then extrapolates and returns the next-hop's interface ID, i.e. the chosen destination neighbor's interface ID. This ID is then passed as a parameter to the `uip_ds6_select_src(..)` function, which computes the sas on the appropriate interface (i.e. on the `uip_ds6_if[interface_id]` interface). This function is illustrated in code listing 3.16.

Source Code 3.16: new function: get next-hop interface ID

```
uint8_t uip_ds6_nbr_get_nexthop_interfaceid(const
    uip_ipaddr_t *dest)
```

It takes as parameter the destination IP address and applies the next-hop determination on it, as the `tcpip_ipv6_output()` does. This function is defined in the uip-ds6-nbr.c source file and has been inserted in correspondence of four sas invocation:

- in uip6.c, in the main function that do the packet processing at network level (IP processing) which is called `uip_process(..)`. This function mainly analyzes the packet header and determines if it's a UDP, TCP or ICMP packet (other header options that are treated by the `uip_process(..)` function are not discussed in this study) and the actions that must be taken; in the three following cases, only unicast addresses are allowed as destination address. TCP and UDP multicast is not implemented, as these protocols are implemented only to establish connections with non-multicast IP addresses
  - during UDP packet sending, in "`udp_send`" case
  - during TCP reset processing, in "reset" case
  - during TCP packet sending, "`tcp_send`" case; code listing 3.17 represents all the three software changes.

  Source Code 3.17: source address selection for a unicast destination in uip6.c

  ```
  #if NETSTACK_CONF_MULTIINTERFACE
  /* Source address selection */
  if (!uip_is_addr_mcast(&UIP_IP_BUF->destipaddr)){
   uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &
      UIP_IP_BUF->destipaddr,
   uip_ds6_nbr_get_nexthop_interfaceid(&UIP_IP_BUF->
      destipaddr));
  ```

```
} else {
 goto drop;
}
#else
uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &
    UIP_IP_BUF->destipaddr);
#endif
```

The new piece of code first check if the destination address that is considered is a multicast address. If it's not the case, then the sas is invoked with the next-hop interface ID parameter extrapolated by the new function, else it throws an error.

- uip-icmp6.c, the source file which handles the ICMP packet processing, in the `uip_icmp6_send(..)` function that sends out a ICMPv6 packet. Here, a multicast destination address is allowed, as this function is exploited by RPL to send out control messages as DIOs. The interface ID in the multicast case can be retrieved from the global `uip_ds6_interfaceid` variable, as illustrated in code listing 3.18.

Source Code 3.18: source address selection for a unicast destination in uip-icmp6.c

```
#if NETSTACK_CONF_MULTIINTERFACE
/* Source address selection */
if (!uip_is_addr_mcast(&UIP_IP_BUF->destipaddr)){
  uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &
     UIP_IP_BUF->destipaddr,
  uip_ds6_nbr_get_nexthop_interfaceid(&UIP_IP_BUF->
     destipaddr));
} else {
uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &UIP_IP_BUF
    ->destipaddr,
  uip_ds6_interface_id);
}
#else
uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &UIP_IP_BUF
    ->destipaddr);
#endif
```

Other instances that do the sas but where the interface ID is easily retrievable, i.e. where the new function has not been exploited, are the following:

- uip-nd6.c, that is the ND6 routing protocol source file:

– NS input, in the `ns_input()` function. In this occurrence, if the target address corresponds to one of the node's addresses, the relative interface ID can be found in the relative field of the address structure retrieved by exploiting the `uip_ds6_addr_lookup()` function, as illustrated in code listing 3.19.

Source Code 3.19: source address selection for a unicast destination in uip-nd6.c during NS input

```
addr = uip_ds6_addr_lookup(&UIP_ND6_NS_BUF->
    tgtipaddr);

[...]

#if NETSTACK_CONF_MULTIINTERFACE
uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &
    UIP_IP_BUF->destipaddr,
uip_ds6_interface_id);
#else
uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &
    UIP_IP_BUF->destipaddr);
#endif
```

– RS output, in the `uip_nd6_rs_output()` function. Here, the `interface_id` is passed as a parameter, as illustrated in code listing 3.20.

Source Code 3.20: source address selection for a unicast destination in uip-nd6.c during RS output

```
void
#if NETSTACK_CONF_MULTIINTERFACE
uip_nd6_rs_output(uint8_t interface_id);
#else
uip_nd6_rs_output(void);
#endif
```

– RA output, in the `uip_nd6_ra_output()` function, as illustrated in code listing 3.21.

Source Code 3.21: Source address selection for a unicast destination in uip-nd6.c during RA output

```
void
#if NETSTACK_CONF_MULTIINTERFACE
uip_nd6_ra_output(uip_ipaddr_t* dest, uint8_t
    interface_id);
#else
uip_nd6_ra_output(uip_ipaddr_t* dest);
#endif
```

Note that the solicited RA in Contiki 3.0 is not implemented, i.e. after a RS input a timer is set and, after it expires, a multicast RA is sent.

- uip-icmp6.c:
  - in the `uip_icmp6_error_output()` function. Here, there are two different cases in which sas is invoked:
    - multicast destination address: if the interface ID is not NULL (actually, is not set to 255 that acts like NULL), the interface ID information is retrievable from the `uip_ds6_interfaceid` variable, i.e. it's ok to use a source address that corresponds to one of the addresses of the interface from which the packet entered. If the `uip_ds6_interfaceid` is NULL, then the packet entered via SLIP with tunslip6. In this case, the sas can be applied on any node's interface, as it makes no difference. Thus, the 0-interface is fine, as illustrated in code listing 3.22.

Source Code 3.22: easy source address selection for multicast destination in uip-icmp6.c

```
#if NETSTACK_CONF_MULTIINTERFACE
/* Source address selection */
if(uip_ds6_interfaceid != 255) {
  /* ICMP packet from a neighbor */
  uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &
     UIP_IP_BUF->destipaddr,
  uip_ds6_interfaceid);
} else {
 /* ICMP packet from an external network, maybe
     through tunslip. This case, even the
    source address of the first interface is
    good. */
 uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &
    UIP_IP_BUF->destipaddr, 0);
}
#else
uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &
   UIP_IP_BUF->destipaddr);
#endif
```

    - unicast destination address: this branch handles different cases, but the worst one for sas complexity is the case in which the source address is a link-layer address and the destination address is off-link, i.e. either belongs to a node that is not directly reachable from the node who received the ICMP packet or doesn't belong to a node at all. In this case, a valid source address for the ICMP error packet must be chosen. A good strategy is to respond to the source address with one address taken from the list of the interface from which the packed entered. Thus, the code block to insert is the same as the previous one: if the `uip_ds6_interfaceid` variable is not NULL (actually, is not set to 255 that acts like NULL), the interface ID information is retrievable from there, i.e. it's ok to use a source address that corresponds to one of the addresses of the interface from which the packet entered. If the `uip_ds6_interfaceid` is NULL, then the packet entered via SLIP with tunslip6. In this case, the sas can be applied on any node's interface, as it makes no difference. Thus, the 0-interface is fine.

– In the `echo_request_input()` function, that handles the ICMPv6 echo requests (i.e. ping6). Here is prepared an echo reply, and there are two different cases to be handled:

  * multicast destination address: if the `uip_ds6_interfaceid` variable is not NULL (actually, is not set to 255 that acts like NULL), the interface ID information is retrievable from there, i.e. it's ok to use a source address that corresponds to one of the addresses of the interface from which the packet entered. If the `uip_ds6_interfaceid` is NULL, then the ping6 entered via SLIP with tunslip6. In this case, the sas can be applied on any node's interface, as it makes no difference. Thus, the 0-interface is fine.
  * unicast destination address: in this case, the sas is not necessary since the source address to utilize in the echo reply is the old destination address of the ping6 packet, as illustrated in code listing 3.23:

Source Code 3.23: easy source address selection for a unicast destination in uip-icmp6.c

```
if(uip_is_addr_mcast(&UIP_IP_BUF->destipaddr)){
  uip_ipaddr_copy(&UIP_IP_BUF->destipaddr, &
      UIP_IP_BUF->srcipaddr);
#if NETSTACK_CONF_MULTIINTERFACE
/* Source address selection */
  if(uip_ds6_interfaceid != 255) {
    /* multicast ping comes from a neighbor */
    uip_ds6_select_src(&UIP_IP_BUF->srcipaddr,
       &UIP_IP_BUF->destipaddr,
    uip_ds6_interfaceid);
  } else {
    /* multicast ping comes from an external
       network (maybe through tunslip), so the
       first interface source address is good.
       */
   uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &
      UIP_IP_BUF->destipaddr, 0);
  }
#else
  uip_ds6_select_src(&UIP_IP_BUF->srcipaddr, &
     UIP_IP_BUF->destipaddr);
#endif
} else {
 uip_ipaddr_copy(&tmp_ipaddr, &UIP_IP_BUF->
    srcipaddr);
 uip_ipaddr_copy(&UIP_IP_BUF->srcipaddr, &
    UIP_IP_BUF->destipaddr);
 uip_ipaddr_copy(&UIP_IP_BUF->destipaddr, &
    tmp_ipaddr);
}
```

RPL UDP Server IP Aliasing

In Contiki, rpl-udp-clients are aware of the rpl-udp-server because the server address is hard-coded in the clients code. This is necessary to let the clients correctly communicate with the server. In a RPL net where the subnets prefixes can change over time, it's important that the server address is prefix independent, as it must be reachable from the clients wherever he is, i.e. in any subnet he is. For this reason, the server address lists must hold at least three unicast addresses:

- one link-local address, obtained from the MAC address
- one global address obtained from the MAC address and from the prefix assigned from the 6LBR. Note that the prefix could change over time
- one prefix independent global address, i.e. not obtained via SLAAC (so it's manually configured)

This implementation is named IP aliasing: it's the association of more than one IP address to a network interface. Note that the prefix independent global address must be hold from every server's interfaces' address lists.

In this work, the chosen hard-coded RPL server IP address is aaaa:abcd::ff:fe00:1. Since this address must not change over time, it must be handled with care by the RPL. The server's addresses lists initialization phase is represented by the code listing 3.24:

Source Code 3.24: RPL UDP server's addresses lists initialization

```
#if NETSTACK_CONF_MULTIINTERFACE
int aaaa = 0xaaaa;
uint8_t i;
for(i=0; i < NETSTACK_CONF_NUMINTERFACES; i++) {
  /* Server's subnet-independent IP has infinite lifetime
     */
  uip_ip6addr(&ipaddr, 0xaaaa, 0xabcd, 0, 0, 0, 0x00ff, 0
     xfe00, 1);
  uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL, i);

  uip_ip6addr(&ipaddr, aaaa+4369*i, 0, 0, 0, 0, 0, 0, 0);
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF, i);
}
#else
uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);
#endif
```

When the RPL detects a new prefix in a received DIO and prepares to delete the old global address in order to substitute it with a new one (that differs from the previous only for the prefix), it must check that the global address is not `ADDR_MANUAL`, since an address configured manually must not be deleted.

Routing Control Packets - Multicast Output

As stated before, multicast packet output are handled in a "for" cycle in order to let all the RPL nodes receive the routing control packet. Hence, a node must send as many multicast packets as the number of the node's interfaces. In the tested working configurations, the 6LBR sends, at every respective timer expiration, two multicast packets: one on the SPIRIT1 channel and the other on the power-line. Instead, the mono-interface nodes send only one multicast packet on the relative channel.

The main multicast packet output cases that occur on RPL nodes are the following:

- DIO multicast output after DIO timer expiration: DIO timer expiration is handled by the `handle_dio_timer()` function in rpl-timers.c
- DIS multicast output:
  - after periodic timer expiration: periodic timer expiration is handled by the `handle_periodic_timer(..)` function in rpl-timers.c
  - in `rpl_dag_root_init_dag()` function, which is invoked by nobody (i.e. noy used)
- DAO multicast output:
  - after DAO timer expiration: DAO timer expiration is handled by the `handle_dao_timer()` function in rpl-timers.c. The DAOs sent in correspondence of this timer expiration contain all the multicast addresses held in the relative multicast address list
  - in the `dao_output(..)` function. There, many DAOs are sent in order to communicate the parent set the addresses that the node holds. Thus, the server node, which holds two global addresses in order to implement the IP aliasing, must send two DAO packets to the neighbors

The main multicast packet output cases that occur on ND6 over 6LoWPAN nodes are the following:

- RS multicast output: invoked by the `uip_ds6_send_rs()` function in uip-ds6.c source file, which is invoked after the RS timer expiration in a non-router node
- RA multicast output: invoked by the `uip_ds6_send_ra_periodic()` function in uip-ds6.c source file, which is invoked after RA timer expiration in a router node

3.3 Software Working Configurations

This section details the example applications that are used in the different tested working configurations.

RPL UDP Border Router

Contiki application flashed on the dual-interface 6LoWPAN Border Router (6LBR) node is the rpl-border-router example, which manages the RPL instance and forwards packets from the nodes that are on the SPIRIT1 radio link to the nodes that are on the ST7580 Powerline link and vice-versa. The example folder path is the following:

```
/examples/ipv6/rpl-border-router/
```

RPL UDP Client

Contiki rpl-udp-client application is flashed on a mono-interface node (without IKS01S01A1 Expansion board) and sends periodically a text message like “Hello -number- via UDP from the client” to a rpl-udp-server node, where -number- is a progressive number. The example can be found in:

```
/examples/ipv6/rpl-udp/
```

RPL UDP Client with Sensors

rpl-udp-client application has been modified to integrate the IKS01A1 Sensors Expansion Board. This new example application captures temperature, pressure, humidity, acceleration, magneto and gyroscope values by reading the sensors in the Expansion Board and then sends the read value to the rpl-udp-server node via SPIRIT1 or ST7580 Powerline Modem. The new example can be found in:

```
/examples/stm32nucleo-spirit1/rpl-udp-client-sensors/
```

RPL UDP Server

Contiki rpl-udp-server application is flashed on a mono-interface node and receives periodically the values read from sensors and/or the hello messages transmitted by the client nodes. The example can be found in:

```
/examples/ipv6/rpl-udp/
```

Alpha Working Configuration

The most interesting working configuration obtained in this work has been called “Alpha” working configuration. The Alpha working configuration has been identified as the configuration that has the minimum number of nodes that are needed in order to demonstrate the new Contiki multi-interface feature, and is composed of:

- one SPIRIT1-ST7580 6LBR node
- one SPIRIT1 RPL-UDP client node with sensors
- one ST7580 RPL-UDP server node

Other Working Configurations

Other working configurations successfully tested in this study are reported below.

One relevant working configuration is composed of the following nodes:

- one dual-interface RPL UDP border router
- some RPL UDP SPIRIT1 Clients
- some RPL UDP Powerline Clients
- some RPL UDP SPIRIT1 Clients with Sensors
- some RPL UDP Powerline Clients with Sensors
- one RPL UDP Powerline Server

Another relevant working configuration is composed of the following nodes:

- one dual-interface RPL UDP border router
- some RPL UDP Powerline Clients
- some RPL UDP SPIRIT1 Clients
- some RPL UDP SPIRIT1 Clients with Sensors
- some RPL UDP Powerline Clients with Sensors
- one RPL UDP SPIRIT1 Server

As can be observed, what remains always the same is the multi-interface RPL border router.

3.4 Software Implementation

This section explains how the example application ELF and binary files can be installed on the boards. In addition, this section explain how to run tunslip6.

The ELF file of a generic Contiki application for a specific platform is obtained by executing the shell command line illustrated in code listing 3.25 in the corresponding example folder.

Source Code 3.25: bash command line for obtaining an ELF file

```
make TARGET=platform_name
```

A platform that includes a SPIRIT1 Expansion Board must use an additional parameter to the shell command line, as illustrated in code listing 3.26

Source Code 3.26: bash command line for obtaining an ELF file for a platform with SPIRIT1

```
make TARGET=platform_name BOARD=board_name
```

where board name can be for instance “ids01a5” or “ids01a4” depending on the SPIRIT1 Expansion Board chosen for the project.

A platform that includes a IKS01A1 Expansion Board must specify also another parameter as illustrated in code listing 3.27.

Source Code 3.27: bash command line for obtaining an ELF file for a platform with sensors

```
make TARGET=platform_name BOARD=board_name
```

In order to install an example application on the Nucleo, the relative ELF file must be converted into a binary (.bin) file with the GNU arm-none-eabi-objcopy utility as illustrated in code listing 3.28.

Source Code 3.28: bash command line for obtaining a binary from an ELF

```
arm-none-eabi-objcopy -O binary elf_file_name bin_file_name
    .bin
```

where "`bin_file_name.bin`" is the name wanted for final binary to be flashed in the Nucleo.

The binary installation can be easily done by copying the .bin file in the device folder after that the board has been connected to the host via USB and (automatically) recognized as a "Mass Storage Device". For instance, the .bin file can be copied as illustrated in code listing 3.29.

Source Code 3.29: copying the bin into the device folder

```
cp bin_file_name.bin /media/user_name/board_factory_name
```

Note that it is recommended to push the Nucleo "reset" button after every flashing operation, as the board could not recognize immediately that a new binary has been flashed.

Tunslip6 for the Border Router

In order to instantiate a RPL net via the tunslip6 utility, the border router must be connected to the host PC. Then, tunslip6 can be invoked as illustrated in code listing 3.30.

Source Code 3.30: instantiate a RPL net via tunslip6

```
sudo ./tunslip6 -i 2 -s /dev/ttyACM0 -r aaaa:abcd::ff:fe00
    :1 aaaa:aaaa::1/64 aaaa:bbbb::1/64
```

where:

- -i 2 is the number of interfaces of the BR
- /dev/ttyACM0 is the serial port by which the BR interfaces with the host
- aaaa:abcd::ff:fe00:1 is the server IP address
- aaaa:aaaa::1/64 is the SPIRIT1 subnet prefix
- aaaa:bbbb::1/64 is the ST7580 subnet prefix

If everything is successful, the BR receives the subnet prefixes and starts to disseminate them via DIOs both on the SPIRIT1 link and on the ST7580 Power Line link.

In order to verify that the BR recognizes its neighbors, i.e. the client nodes and the server node, the BR web-page can be used. It can be accessed by opening in a web browser one of the two BR IP addresses. For instance:

```
[aaaa:aaaa::500:f8ff:2ec6:d139]
```

or

```
[aaaa:bbbb::500:f8ff:2ec6:d139]
```

The web page lists the neighbors' link-local addresses under the "Neighbors" title, and global addresses reachable via the neighbors under the "Routes" title.

Chapter 4

Evaluation

This Chapter presents some memory size and performance analysis made on testing configurations presented in Chapter 3.

Goal of these analysis is to evaluate the impact that the addition of the multi-interface feature has on the already existing Contiki system from the total memory occupation and from the runtime execution point of views.

4.1 Memory Footprint

In an ARM ELF file, there are three types of Segment:

- Text: contains the code for the executable. With respect to this thesis work, text contains what ends up in flash memory. It contains functions and constant data. It contains also the interrupt vector table
- Data: contains initialized read-write data for the executable. Initialized data are not constants, so they will end up in RAM. However, data initialization values are constants, thus live in flash memory. The initialization of the variable is done during the normal startup code. Thus, when considering an object file size, data size must be counted twice: data size occupied both in RAM and in flash/ROM
- BSS: contains uninitialized data, which should be zeroed either when an image is created, or at program startup by the runtime environment. BSS ends up in RAM

The memory footprint of an executable program indicates its memory requirements. This includes all kind of memory regions that the program ever needs while executing and will be loaded at least once during the entire run, i.e. text, data and BSS segments.

The measured memory footprint is represented in kilobytes, where 1 kilobyte = 1000 bytes as stated by the International System of Units (SI).

The memory footprint of an executable can be obtained with the "size" GNU ARM toolchain utility on its relative ELF file. The version of the ARM Toolchain (GNU Tools for ARM Embedded Processors) used in this work is 2.24.0.20150921. Next sections illustrate the memory footprint for the main Contiki applications tested on the working configurations.

The memory footprint of a Contiki application can be obtained with the bash command line illustrated in code listing 4.1.

Source Code 4.1: command line to obtain memory footprint

```
arm-none-eabi-size application_name
```

where `application_name` is the name of the application (for instance, border-router).

RPL Border Router ELF Footprint

Memory footprint of a multi-interface RPL border router grows linearly with the number of the interfaces managed by the node. Moreover, a fixed memory overhead that does not change with the number of interfaces is caused by source-code changes. Thus, the text segment is independent from the number of interfaces of the node, and it is fixed. Instead, data and BSS segments change linearly with the number of interfaces.

Memory footprint has been measured on a dual-interface (ST7580 Powerline and SPIRIT1) BR. Sizes, expressed in bytes, are illustrated in table 4.1 on the next page, where:

- "dec" represents the total memory occupation expressed in decimal representation, thus it is the sum of the text, data and bss sections sizes, i.e. 146088 bytes ∼146 kilobytes
- "hex" is the hexadecimal representation of the total memory occupation, thus 146088 bytes in decimal equals to 23aa8 in hexadecimal.

Table 4.1: RPL dual-interface border router memory sections sizes

text	data	bss	dec	hex
133624	2904	9560	146088	23aa8

BSS and data sections of a 3-interface and 4-interface BR have been measured as well in order to verify if a costant overhead due to the addition of the n+1 interface can be found. Sections sizes are reported in table 4.2 on the following page. The third and the fourth interfaces have been obtained by duplicating the SPIRIT1 radio Netstack drivers and the ST7580 powerline Netstack drivers, as illustrated in figures 4.1 and 4.2.

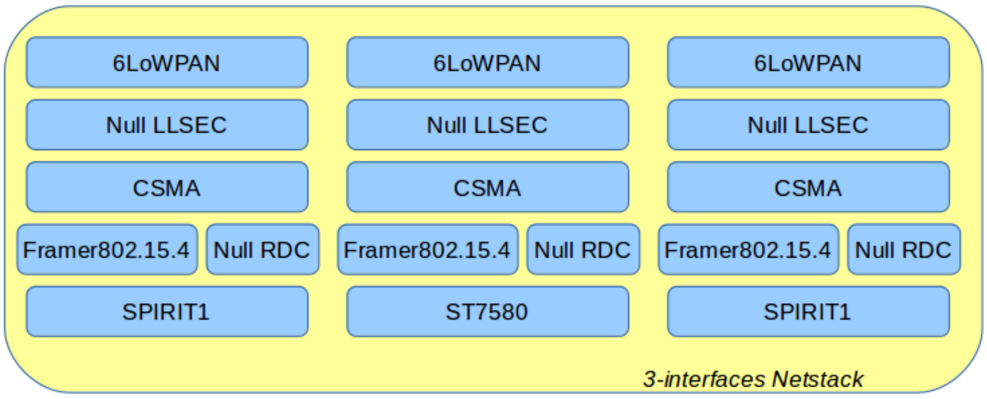


Figure 4.1: 3-interfaces device for memory sizes analysis

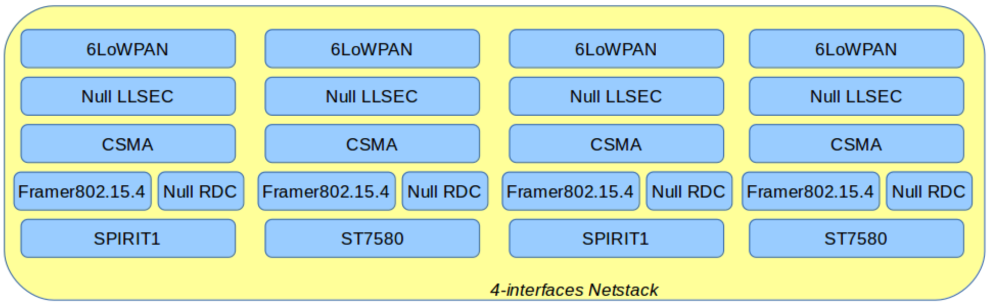


Figure 4.2: 4-interfaces device for memory sizes analysis

Memory footprint of the original Contiki RPL border router example, i.e. without multi-interface code modifications, has been obtained by considering the worst sections measurements taken on both the Nucleo+SPIRIT1 and the Nucleo+ST7580 configurations in order to make a

Table 4.2: n-interfaces device RAM sections sizes

	data (bytes)	bss (bytes)	RAM (bytes)
3-interface BR	10120	2904	13024
4-interface BR	10680	2904	13584

Table 4.3: original RPL border router memory sections sizes (bytes)

text	data	bss	dec	hex
122680	2880	8536	132860	206fc

fair comparison with the dual-interface implementation, that incorporates both configurations. Values are illustrated in table 4.3.

The following comparison tables allow to easily compare the two Contiki BR implementations. The overhead value of a section is obtained by subtracting the original Contiki BR implementation value to the multi-interface implementation value. The FLASH value of an implementation is obtained by adding the text value to the data value of the same row. FLASH values are reported in table 4.4 The RAM value of an implementation is obtained by adding the BSS value to the Data value of the same row. RAM values are reported in table 4.5 on the next page and 4.6 on the facing page.

As can be observed, the FLASH overhead is about 11 KB and the RAM overhead is about 1 KB.

Overhead by adding one interface to a device which is already utilizing a multi-interface application (i.e. not an original Contiki application) is costant and is equal to 560byte. This value has been verified to be approximately induced from the vectorized variables, i.e. originally not arrays, illustrated in table 4.7 on the next page.

Percentages are obtained by comparing the dual-interface BR memory overhead with the original Contiki BR memory footprint. FLASH overhead is 8,74% while RAM overhead is 10,71%.

Table 4.4: border router FLASH sizes comparison (bytes)

	text	data	FLASH
Original Contiki BR	122680	2880	125560
Dual-Interface BR	133624	2904	136528
Overhead (difference)	10944	24	10968

Table 4.5: border router RAM sizes comparison (bytes)

	BSS	data	RAM
Original Contiki BR	8536	2880	11416
Dual-Interface BR	9560	2904	12464
Overhead (difference)	1024	24	1048

Table 4.6: multi-interface BR RAM sizes comparison (bytes)

	BSS	data	RAM
Dual-Interface BR	9560	2904	12464
3-interface BR	10120	2904	13024
4-interface BR	10680	2904	13584

Table 4.7: vectorized variables that greatly influence RAM overhead

Vectorized variable	Size (Byte)
`uip_ds6_if[NETSTACK_NUM_INTERFACES]`	260
`NETSTACK_NETWORK[..]`	12
`NETSTACK_LLSEC[..]`	16
`NETSTACK_MAC`	28
`NETSTACK_RDC`	32
`NETSTACK_FRAMER`	12
`NETSTACK_RADIO`	56
`uip_ds6_prefix_list[..]`	96
dag->`prefix_info[..]`	24
Total	536

Table 4.8: RPL UDP mono-interface client memory sections sizes

text	data	bss	dec	hex
124268	2840	7828	134936	20f18

Table 4.9: original RPL UDP client memory sections sizes

text	data	bss	dec	hex
117496	2840	7636	127972	1f3e4

RPL UDP Client ELF Footprint

In order to verify that the RAM memory footprint does not change for a mono-interface application with respect to an original application, measurements have been done by compiling a rpl-udp client application respectively by declaring one interface in the new multi-interface environment in the first case and by disabling the multi-interface environment in the second case. In other words, new software changes contained in "if" directive code blocks have been enabled in the first case together with the declaration of a number of interfaces equal to one, while in the second case new software changes have not been enabled by declaring that the environment must not be multi-interface.

The resulting memory occupation for the SPIRIT1-interface Client example is illustrated in table 4.8.

The memory footprint of the original Contiki RPL UDP Client example, i.e. without multi-interface code modifications, is illustrated in table 4.9.

Comparison tables are reported below.

As can be observed, FLASH overhead is about 7 KB and it is caused by the source-code changes. RAM overhead is in the order of bytes and it is caused by the BSS segment. As expected, mono-interface client RAM

Table 4.10: RPL UDP client FLASH sizes comparison (bytes)

	text	data	FLASH
Original Contiki client	117496	2840	120336
Mono-interface client	124268	2840	127108
Overhead	6772	0	6772

Table 4.11: RPL UDP client RAM sizes comparison (bytes)

	BSS	data	RAM
Original Contiki client	7636	2840	10476
Mono-interface client	7828	2840	10668
Overhead	192	0	192

Table 4.12: RPL UDP mono-interface client with sensors memory sections sizes

text	data	bss	dec	hex
152740	3312	7928	163980	2808c

overhead is approximatively null. FLASH overhead is 5,63% while RAM overhead is 1,84%.

RPL UDP Client with Sensors ELF Footprint

This application example was introduced with this work, hence it was not furnished with the Contiki application development kit. Anyway, it can be referred to as “Original Contiki Client” when the `NETSTACK_CONF_MULTIINTERFACE` macro is set to 0, that is the (multi-interface) software changes introduced with this work are disabled.

The resulting occupation for the mono-interface Client with sensors example is illustrated in table 4.12.

The memory footprint of the RPL UDP Client with sensors example without multi-interface code modifications (i.e. `NETSTACK_CONF_MULTIINTERFACE` macro set to 0) is illustrated in table 4.13.

Comparison tables are reported below.

As can be observed, FLASH overhead is about 4 KB and RAM overhead is in the order of bytes. FLASH overhead is 2,46% while RAM overhead is

Table 4.13: RPL UDP client with sensors without multi-interface changes memory sections sizes

text	data	bss	dec	hex
149012	3304	7736	160052	27134

Table 4.14: RPL UDP client with sensors FLASH sizes comparison (bytes)

	text	data	FLASH
Original Contiki client	149012	3304	152316
Mono-interface client	152740	3312	156052
Overhead	3728	8	3736

Table 4.15: RPL UDP client with sensors RAM sizes comparison (bytes)

	BSS	data	RAM
Original Contiki client	7736	3304	11040
Mono-interface client	7928	3312	11240
Overhead	192	8	200

1,82%.

RPL UDP Server ELF Footprint

In order to verify that the RAM memory footprint does not change for a mono-interface node, measurements have been done on a mono-interface node by compiling a rpl-udp server application respectively with the software changes (declaring only one interface) and without them.

Memory footprint measurement has been done on the mono-interface Server exploited in the alpha working configuration, i.e. a ST7580-based Server.

The resulting information is illustrated in table 4.16.

Comparison tables are reported below.

As can be observed, FLASH overhead is about 4 KB and RAM overhead is in the order of bytes. As expected, mono-interface server RAM overhead is approximatively null. FLASH overhead is 3,93% while RAM overhead is 1,75%.

Table 4.16: RPL UDP mono-interface server memory sections sizes (bytes)

text	data	bss	dec	hex
92136	2632	9016	103784	19568

Table 4.17: RPL UDP server FLASH sizes comparison (bytes)

	text	data	FLASH
Original Contiki client	88564	2624	91188
Mono-interface client	92136	2632	94768
Overhead	3572	8	3580

Table 4.18: RPL UDP server RAM sizes comparison (bytes)

	BSS	data	RAM
Original Contiki client	8824	2624	11448
Mono-interface client	9016	2632	11648
Overhead	192	8	200

4.2 Performance

In order to evaluate the system performance at runtime, system clock cycles have been measured on a border router (both in the multi-interface version and in the original mono-interface version) in the following cases:

- while a packet cross the Netstack from radio driver to RPL module and viceversa (section "Packet Netstack Crossing Performance")
- during a DIO multicast output invocation, i.e. exit from a for cycle where the `dio_output(..)` function is invoked as many times as the number of interfaces the node has (section "Multicast DIO Output Performance")
- during a RPL instance initialization (section "BR RPL Instance Initialization Performance")

These test cases were chosen because they cover the software changes apported by this thesis work.

Number of interfaces of a multi-interface implementation must be declared in the contiki-conf.h file as follows:

```
#define NETSTACK_CONF_NUMINTERFACES num_interfaces
```

In order to compare the multi-interface implementation with the original one, three configurations have been analyzed:

- the dual-interface border router: macro has been configured to 2

- the mono-interface border router: macro has been configured to 1
- the original Contiki border router: the macro doesn't exist in this configuration, as it as been introduced with the multi-interface implementation obtained during this work

In order to measure the system clock cycles, a ARM -M3 core debug register named "cycle counter register", which is part of the "data watchpoint trigger" registers set, has been used. Counter definition can be found in:

```
cpu/arm/common/CMSIS/core_cm3.h
```

Functions for modifying such registers and to read the counter have been implemented in dwt.c and dwt.h files, that can be found in appendix section ARM Cortex Cycle Counter Handling.

Counter is set by default at 32 MHz. When the counter is enabled, it starts counting the system clock cycles until it is disabled again. The counter must be initialized in Contiki main (respectively in contiki-spirit1-st7580-main.c, in contiki-spirit1-main.c and in contiki-st7580-main.c source files).

4.2.1 Packet Netstack Crossing Performance

This test measures the clock cycles that a border router needs to allow a packet to cross the Netstack from radio driver to RPL and viceversa. This I/O path covers all the software changes made in the code, as it includes the routing logic and routing tables management.

From the hardware point of view there are two test configurations:

- NucleoL152RE with `X_Nucleo_IDS01A5`
- NucleoL152RE with `X_Nucleo_ST7580`

Note that the time required to cross the Netstack during a packet I/Os in a multi-interface BR implementation is not affected by the number of interfaces of the node, as every Netstack array is accessed in a fixed time by using the interface ID information as array index. Hence, a mono-interface BR which implements the multi-interface feature can be used to make a performance comparison with the original Contiki implementation.

There are two main operations to test:

- Packet input: the counter is started at physical layer and stopped at routing level. `NETSTACK_RDC.input()` call can be taken as the "start reference" for cycle measure and DIO input or DIS input function call can be taken as "stop reference". In order to compare the results, only not fragmented input packets are registered. Clock counter is handled by the following functions. Note that only SPIRIT1 functions (and not ST7580 Powerline ones) are reported as the mechanism is exacltly the same for ST7580 functions.

  – `PROCESS_THREAD(spirit_radio_process`, ev, data):

  ```
  #endif /* NULLRDC_CONF_802154_AUTOACK */

    dwt_enable();

    packetbuf_set_datalen(len);
    NETSTACK_RDC.input();
  ```

  in the original Contiki SPIRIT1 driver and:

  ```
  dwt_enable();

  packetbuf_set_attr(PACKETBUF_ATTR_INTERFACEID,
      SPIRIT1_INTERFACEID);
  packetbuf_set_datalen(len);
  NETSTACK_RDC[SPIRIT1_INTERFACEID].input();
  ```

  in the multi-interface platform SPIRIT1 driver

  – `dio_input`(..):

  ```
  static void
  dio_input(void)
  {
    cycles = dwt_get_cycles();
    printf("DIO Input. System cycles: %u\n", cycles)
        ;
    dwt_disable();
  ```

  in both the original Contiki RPL code and in the new multi-interface implementation.

  – `dis_input`():

  ```
  static void
  dis_input(void)
  {
    cycles = dwt_get_cycles();
    printf("DIS Input. System cycles: %u\n", cycles)
        ;
  ```

```
  dwt_disable();
```

in both the original Contiki RPL code, and in the new multi-interface implementation.

- Packet output: the counter is started at routing level and stopped at physical layer. DIO output function call can be taken as the "start reference" for cycle measure and `spirit_radio_send` function call can be taken as "stop reference".

  - `spirit_radio_send`(...):

```
static int
spirit_radio_send(const void *payload, unsigned
   short payload_len)
{
  if(dio_out) {
    cycles = dwt_get_cycles();
    printf("DIO OUTPUT. System cycles: %u.\n",
       cycles);
    dwt_disable();
    dio_out=0;
  }
```

  in both the original and the multi-interface Contiki SPIRIT1 driver, when measuring the DIO output cycles;

  - `handle_dio_timer`(..):

```
#endif /* RPL_CONF_STATS */

  dio_out=1;
  dwt_enable();
  dio_output(instance, NULL);
```

  in the original Contiki RPL code, and:

```
#endif /* RPL_CONF_STATS */
#if NETSTACK_CONF_MULTIINTERFACE
  uint8_t i;
  for(i=0; i < NETSTACK_CONF_NUMINTERFACES; i++) {
    dio_out=1;
    dwt_enable();
    dio_output(instance, NULL, i);
  }
#else
  dio_output(instance, NULL);
#endif
```

  in the new multi-interface Contiki implementation.

Table 4.19: Nucleo with SPIRIT1 Performances Comparison

	DIS input		DIO input		DIO output	
	cycles	μs	cycles	μs	cycles	μs
Original OS	5478,55	171,21	7282,6	227,59	15887,35	496,48
Multi-interface	5855,75	183	7631,1	238,48	16268,75	508,4
Extra overhead	377,2	11,79	348,5	10,89	381,4	11,92

Table 4.20: Nucleo with ST7580 Performances Comparison

	DIS input		DIO input		DIO output	
	cycles	μs	cycles	μs	cycles	μs
Original OS	5489,65	171,56	7265	227,04	15956	498,63
Multi-interface	5790,15	180,95	7572,4	236,64	16296,35	509,26
Extra overhead	300,5	9,39	307,4	9,6	340,35	10,64

Measurements on Nucleo with SPIRIT1

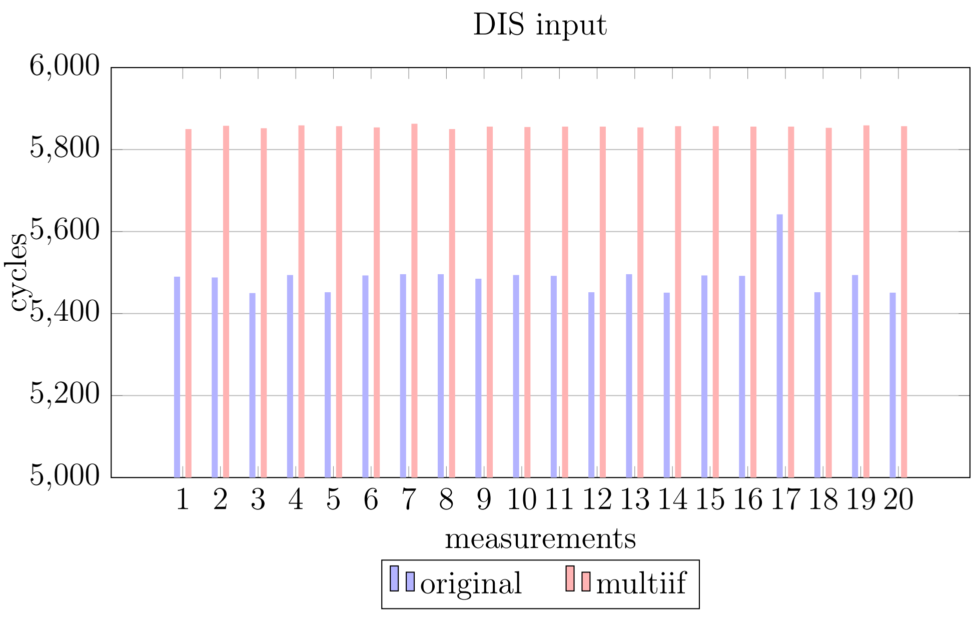
Figure 4.3 on the following page contains graphs which compare respectively DIS input, DIO input and DIO output performances (measured in clock cycles) of an original and a dual-interface BR implementation.

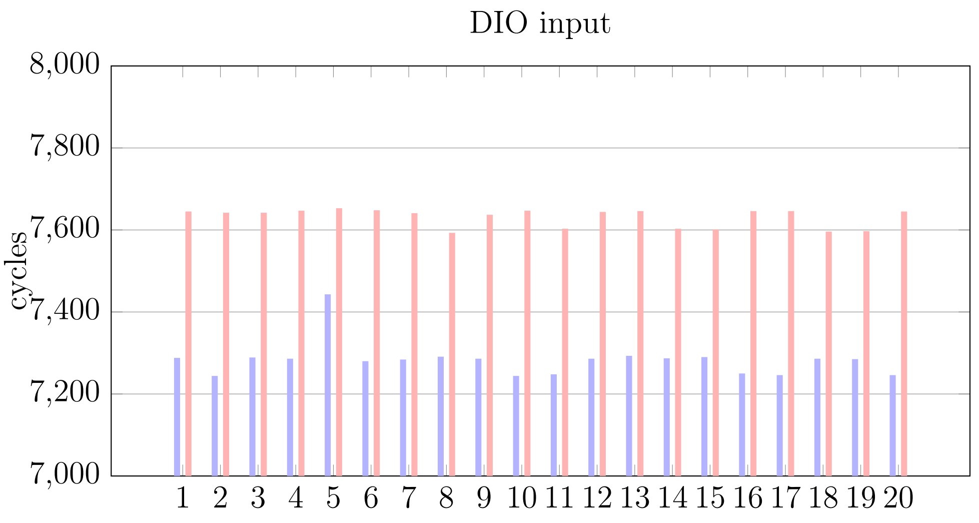
The average number of cycles per packet type has been obtained after collecting twenty measurements. The average processing time per packet type in order to cross the Netstack in microseconds has been obtained by dividing 32 million (which corresponds to Cortex M3's clock speed) to the average number of cycles. Averages are reported in table 4.19.

Measurements on Nucleo with ST7580

Figure 4.4 on page 163, contains graphs which compare respectively DIS input, DIO input and DIO output performances (measured in clock cycles) of an original and a dual-interface BR implementation.

Averages are reported in table 4.20.





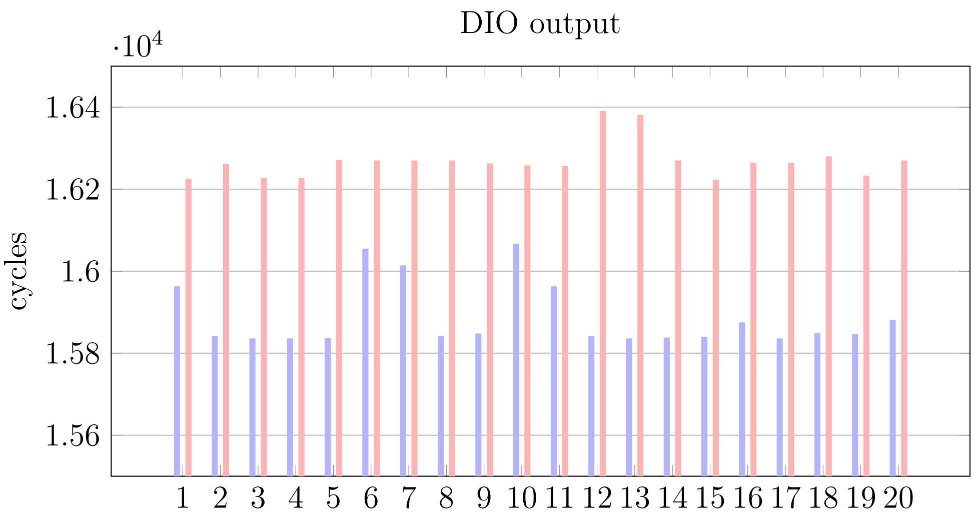


Figure 4.3: original and dual-interface BR performances with SPIRIT1

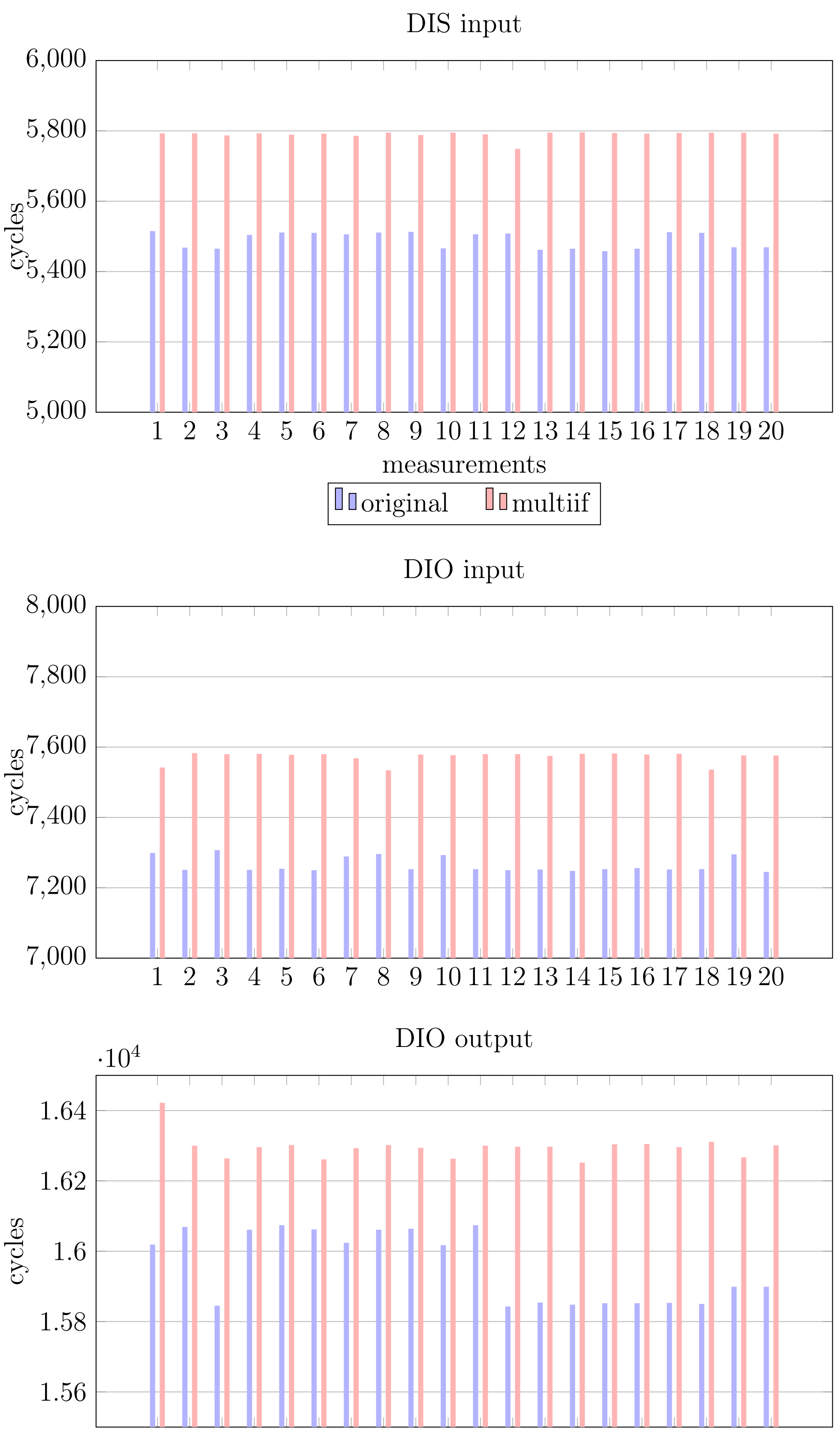


Figure 4.4: original and dual-interface BR performances with ST7580

Table 4.21: multi-interface feature performances comparison

	DIS input		DIO input		DIO output	
	extra cycles	μs	extra cycles	μs	extra cycles	μs
Nucleo+SPIRIT1	377,2	6,89	348,5	4,79	381,4	2,4
Nucleo+ST7580	300,5	5,48	307,4	4,24	340,35	2,14

Multi-interface Configurations Performances Comparison

Table 4.21 illustrates the overheads percentages calculated with respect to the original Contiki implementation.

As can be observed, extra overhead required to cross the Netstack during packet I/Os with the new multi-interface feature is very low, i.e. approximatively:

- 6% for DIS input
- 5% for DIO input
- 3% for DIO output

4.2.2 Multicast DIO Output Performance

This test measures the clock cycles that a border router implementation needs (both in the multi-interface version and in the original mono-interface version) to complete a DIO multicast output invocation, i.e. exit from a for cycle where the `dio_output(..)` function is invoked as many times as the number of interfaces the node has. Every `dio_output(..)` invocation ends in the CSMA driver, when the DIO packet is queued. Thus, the measurements will output a value that represents the clock cycles that the BR implementation needs in order to complete such invocation, not the cycles that needs in order to effectively send a DIO.

The expected dual-interface BR results are values that approximatively double the original Contiki mono-interface BR values, which in turn approximatively match the new mono-interface BR (i.e. multi-interface environment with number of interfaces equal to one) values.

Table 4.22: multicast DIO output performances averages

	Original Contiki BR	Mono-interface BR	Dual-interface BR
Average cycles	9939,3	10162,6	19605,75

Since this test purpose is only to verify that the thrown values are correlated (i.e. multiples), measurements have been taken only on a Nucleo with a SPIRIT1 Expansion Board.

Measurements are reported in graph 4.5.

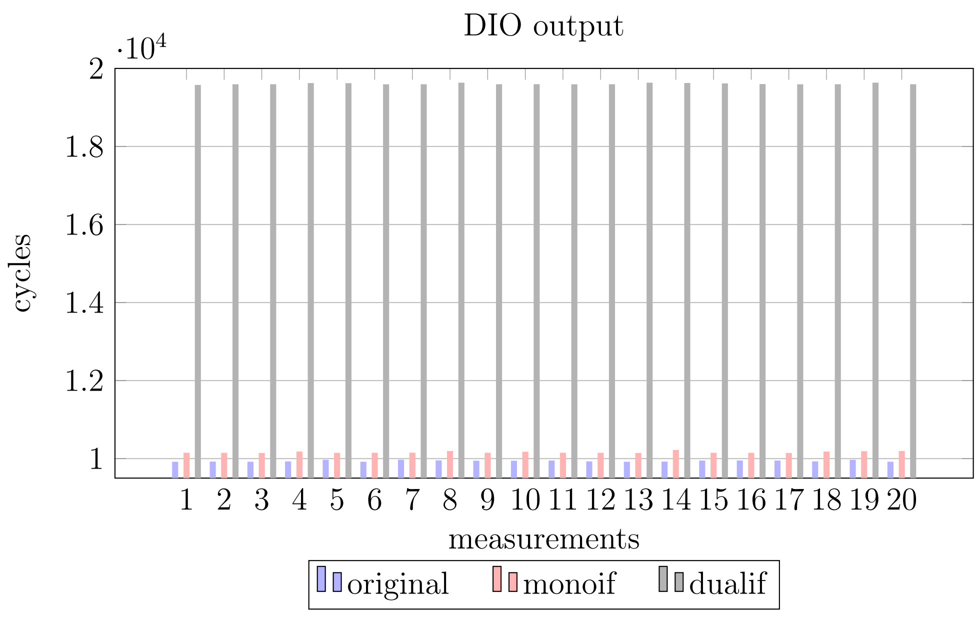


Figure 4.5: multicast DIO output performances comparison

The averages of the previous measurements are reported in table 4.22.

Obtained results show the expected linearity.

4.2.3 BR RPL Instance Initialization Performance

This test analysis measures the clock cycles that a border router implementation needs in order to initialize a RPL istance, i.e. execute the `set_prefix_64(..)` function.

The expected dual-interface BR results are values that approximatively double the original Contiki mono-interface BR values, which in turn approximatively match the new mono-interface BR (i.e. multi-interface environment with number of interfaces equal to one) values.

Counter cycles for the dual-interface and mono-interface BR are analyzed as illustrated by code listing 4.2.

Source Code 4.2: multi-interface BR RPl instance initialization with cycle counter

```
void
set_prefix_64(uip_ipaddr_t *prefix_64)
{
  rpl_dag_t *dag;
  uip_ipaddr_t ipaddr;
#if NETSTACK_CONF_MULTIINTERFACE
  /* Set DAG_ID; in this implementation, it's ok to use the
      first
   * prefix that the SLIP sent us.
   */

  dwt_enable();

  memcpy(&ipaddr, &prefix_64[0], 16);
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  dag = rpl_set_root(RPL_DEFAULT_INSTANCE, &ipaddr);
  if(dag != NULL) {
    PRINTF("BR: Created a new RPL dag with DODAG ID: ");
    PRINT6ADDR(&ipaddr);
    PRINTF("\n");

    /* Set IP addresses to interfaces */
    uint8_t i;
    for(i=0; i < NETSTACK_CONF_NUMINTERFACES; i++) {
      memcpy(&prefix, &prefix_64[i], 16);
      memcpy(&ipaddr, &prefix_64[i], 16);
      uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
      rpl_set_prefix(dag, &prefix, 64, i);
      PRINTF("BR: Interface identifier of link number %d :
          ", i);
      PRINT6ADDR(&ipaddr);
      PRINTF("\n");
      uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF, i);
      prefix_set = 1;
    }
  }

  cycles = dwt_get_cycles();
```

Table 4.23: BR RPL instance initialization performances averages

	Original Contiki BR	Mono-interface BR	Dual-interface BR
Average cycles	5168,2	5975,45	9657,75

```
  printf("BR: RPL instance created. System cycles: %u\n",
     cycles);
  dwt_disable();
```

Counter cycles for the original BR are analyzed as illustrated by code listing 4.3.

Source Code 4.3: original BR RPl instance initialization with cycle counter

```
void
set_prefix_64(uip_ipaddr_t *prefix_64)
{
  rpl_dag_t *dag;
  uip_ipaddr_t ipaddr;
  dwt_enable();

  memcpy(&prefix, prefix_64, 16);
  memcpy(&ipaddr, prefix_64, 16);
  prefix_set = 1;
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);

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

  cycles = dwt_get_cycles();
  printf("BR: RPL instance created. System cycles: %u\n",
     cycles);
  dwt_disable();
}
```

Measurements are reported in graph 4.6 on the following page.

The averages of the previous measurements are reported in table 4.23.

Expected results were obtained, where the mono-interface BR adds a little overhead with respect to the original Contiki code and the dual-interface BR doubles the original BR.

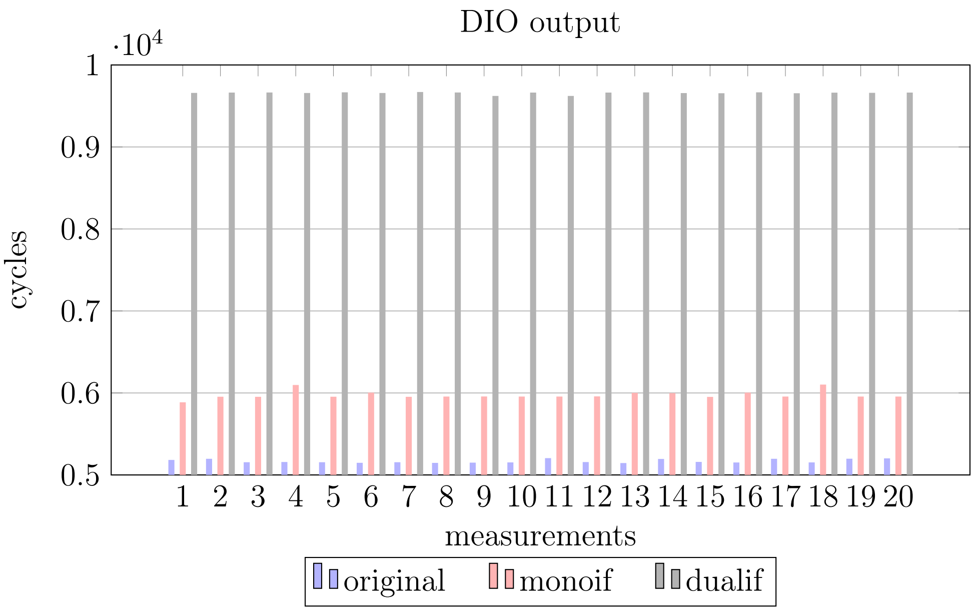


Figure 4.6: BR RPL Instance Initialization Performances

Conclusions

After the evaluation tests carried out, some conclusions can be made.

First of all, the initial main goal of this thesis work has been reached, obtaining a multi interface network prototype compliant with IETF 6LoWPAN, ContikiRPL and IEEE 802.15.4. The prototype has been tested in a real environment with nodes configured according to different working configurations all based on a real hardware platform (STM32L152) and on two real communication interfaces (SPIRIT1 Sub 1Ghz transceiver and ST7580 Power Line Modem).

The new multi-interface feature allows scalability with respect to the number of a RPL border router's interfaces without considerably increasing its memory footprint, since it adds:

- $\sim$9% FLASH overhead (on a total amount of $\sim$126 Kilobytes) for the dual-interface 6LBR implementation and $\sim$5% FLASH overhead (out of $\sim$106 Kilobytes) for RPL UDP Client and RPL UDP Server nodes. FLASH overhead is independent from the number of interfaces utilized by a border router

- $\sim$11% RAM overhead (on a total amount of $\sim$11 Kilobytes) for the dual-interface 6LBR implementation and $\sim$2% RAM overhead (out of $\sim$11 Kilobytes) for RPL UDP Client and RPL UDP Server nodes. In general, RAM overhead depends from the number of interfaces utilized by a border router. This dependency comes out as a $\sim$560 byte RAM overhead for every new interface added to a border router which already implements a multi-interface environment (i.e. which is not utilizing an original Contiki application), i.e $O(1)$

The new feature does not affect considerably system performances, as it adds (independently from the number of interfaces utilized by a border router) $\sim$4,7% overhead to data traffic handling performances (see Evaluation chapter for further details).

As expected, some marginal cases such as multicast outputs and RPL instance initialization reflect performance linearity with respect to the number of interfaces, i.e. ϑ(n).

Future Work

Testing of a 6LoWPAN border router with more than two interfaces has surely the highest priority in future work. Testing of the multi-interface feature with other Contiki examples is also relevant. In this latter case, source code changes should be done in order to adapt the examples to the new feature.

It could be interesting to conceive a network composed by many multi-interface nodes, i.e. a network where not only the border router is equipped with more than one interface. One already identified problem with respect to this conception is related to network subnetting: messages which carry prefix information for stateless address autoconfiguration should be smartly handled by nodes which don't know what subnet they should belong to. Thus, messages propagation will need to be analyzed carefully in order to let all the nodes receive the correct information. A secondary identified problem is when two nodes with both more than one interface want to communicate: they have to choose which is the preferred channel to utilize and, for this purpose, some specific RPL optimization metric could be investigated in addition to those already existing in Contiki.

Appendix A

Appendix

A.1 Border-Router GPIOs Configuration

ST7580 Powerline's GPIO default configuration can be found in:

```
contiki/platform/stm32nucleo-st7580/stm32cube-lib/drivers/
    x_nucleo_st7580/plm_gpio.h
```

and in:

```
contiki/platform/stm32nucleo-st7580/stm32cube-lib/drivers/
    x_nucleo_st7580/plm_uart.h
```

SPIRIT1 Expansion Board's GPIO default configuration can be found in:

```
contiki/platform/stm32nucleo-spirit1/stm32cube-lib/drivers/
    x_nucleo_ids01ax/radio_gpio.h
```

and in:

```
contiki/platform/stm32nucleo-spirit1/stm32cube-lib/drivers/
    x_nucleo_ids01ax/radio_spi.h
```

On the 6LBR, which exploits both the ST7580 Powerline and the SPIRIT1 Expansion Board but does not exploit the sensors Expansion Board, original GPIOs configuration was as illustrated in table A.1 on the following page.

As can be seen, GPIO conflicts were on:

Table A.1: GPIO conflicts

PIN	MACRO	DEVICE
PA0	`USB_DISCONNECT`	nucleo
PA1		
PA2	`USARTx_TX_PIN`	nucleo (SLIP)
PA3	`USARTx_RX_PIN`	nucleo (SLIP)
PA5	`LED2_PIN` (green)	nucleo
PA6	`RADIO_SPI_MISO_PIN`	spirit1
	`PLM_GPIO_RESETN_PIN`	st7580
PA7	`RADIO_SPI_MOSI_PIN`	spirit1
	`PLM_GPIO_T_REQ_PIN`	st7580
PA8	`MCO1_PIN` (microcontroller clock output), used to output SYSCLK, HSI, LSI, MSI, LSE, HSE or PLL clock.	nucleo
PA9	`USARTplm_TX_PIN`	st7580
PA10	`USARTplm_RX_PIN`	st7580
	`RADIO_GPIO_SDN_PIN`	spirit1
PB3	`RADIO_SPI_SCK_PIN`	spirit1
	`PLM_PL_TX_ON_PIN`	st7580
PB4	`LED1_PIN` (red)	spirit1
PB5	`PLM_PL_RX_ON_PIN`	st7580
PB6	`RADIO_SPI_CS_PIN`	spirit1
PC13	`USER_BUTTON_PIN`	nucleo

- PA6: SPIRIT1's SPI MISO signal (data from SPIRIT1 to MCU) versus ST7580 Powerline system reset signal (digital input)

- PA7: SPIRIT1's SPI MOSI signal (data from MCU to SPIRIT1) versus ST7580 Powerline's UART communication control line (digital input)

- PA10: SPIRIT1's shutdown input signal versus ST7580 Powerline's reception analog input

- PB3: SPIRIT1's SPI bit clock versus ST7580 Powerline's transmission in progress output (digital output)

In this study, the ST7580 Powerline was chosen to be connected to the Nucleo board via wires. Anyway, the choice of which expansion board to plug onto the Nucleo board and which one keep unplugged, i.e. connected via wires, is indifferent.

The final 6LBR GPIO configuration is illustrated in table A.2 on the next page.

This GPIO configuration is set in the new Contiki hw-config.h header file, which can be found in the new multi-interface platform folder.

A.2 ARM Cortex Cycle Counter Handling

This section illustrates the content of dwt.h header file and dwt.c source file, which handle the ARM Cortex cycle counter used to evaluate system performances.

Table A.2: GPIO configuration

PIN	MACRO	DISPLACEMENT	DEVICE
PA0	`USB_DISCONNECT`		nucleo
PA1	`PLM_GPIO_RESETN_PIN`	(from PA6)	st7580
PA2	`USARTx_TX_PIN`		nucleo (SLIP)
PA3	`USARTx_RX_PIN`		nucleo (SLIP)
PA4	`PLM_GPIO_T_REQ_PIN`	(from PA7)	st7580
PA5	`LED2_PIN` (green)		nucleo
PA6	`RADIO_SPI_MISO_PIN`		spirit1
PA7	`RADIO_SPI_MOSI_PIN`		spirit1
PA8	`MCO1_PIN` (microcontroller clock output), used to output SYSCLK, HSI, LSI, MSI, LSE, HSE or PLL clock.		nucleo
PA10	`RADIO_GPIO_SDN_PIN`		spirit1
PB0	`PLM_PL_TX_ON_PIN`	(from PB3)	st7580
PB3	`RADIO_SPI_SCK_PIN`		spirit1
PB4	`LED1_PIN` (red)		spirit1
PB5	`PLM_PL_RX_ON_PIN`		st7580
PB6	`RADIO_SPI_CS_PIN`		spirit1
PB10	`USARTplm_TX_PIN`	(from PA9)	st7580
PB11	`USARTplm_RX_PIN`	(from PA10, on morpho)	st7580
PC7	`RADIO_GPIO3_PIN`		spirit1
PC13	`USER_BUTTON_PIN`		nucleo

Source Code A.1: dwt.h

```c
void dwt_init(void);
void dwt_reset(void);
void dwt_disable(void);
void dwt_enable(void);
unsigned int dwt_get_cycles(void);
```

Source Code A.2: dwt.c

```c
#include "spirit1.h"
#include "dwt.h"


void dwt_init(void)
{
  CoreDebug->DEMCR |= (1 << 24); //Enable
      TRCENA bit for using DWT
}

void dwt_disable(void)
{
  DWT->CTRL |= (0 << 0);
}

void dwt_reset(void)
{
  DWT->CYCCNT = 0;
}

void dwt_enable(void)
{
  dwt_reset();
  DWT->CTRL |= (1 << 0);
}

unsigned int dwt_get_cycles(void)
{
  return DWT->CYCCNT;
}
```

A.3 Debugging the Alpha Configuration

This section illustrates how to retrieve the debug messages sent by the client and the server via serial line.

Contiki Source Files Debug

In order to see the "PRINTF(..)" messages, which are exploited during debug, it's enough to change the following code line retrievable in the source files:

```
#define DEBUG DEBUG_NONE
```

with this line:

```
#define DEBUG DEBUG_PRINT
```

or, identically, to change the following line:

```
#define DEBUG 0
```

with this line:

```
#define DEBUG 1
```

Minicom for the Server

In order to retrieve the server debug messages, a serial line connection must be established between the server node and the host. Minicom is a terminal emulation program for the serial line that can be run by typing the following shell command line:

```
sudo minicom --setup
```

The setup option opens a menu where can be found the "serial port setup" option. This option allows to select the correct serial port via which the server connects. If the server was the second device to be connected, then the /dev/ttyACM1 must be selected.

In order to display correctly the debug messages, type: "Ctrl+A" and then type "Z".

Minicom for the Client

The same operation must be done for the Client, where the serial port must be configured as /dev/ttyACM2.

Acronyms

6LBR	6LoWPAN Border-Router
6LoWPAN	IPv6 over LoWPAN
BR	Border-Router
BSS	Block Started by Symbol
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
DAG	Directed Acyclic Graph
DAO	Destination Advertisement Object
DIO	DODAG Information Object
DIS	DODAG Information Solicitation
DODAG	Destination Oriented Directed Acyclic Graph
ELF	Executable and Linkable Format
IETF	Internet Engineering Task Force
IoT	Internet of Things
ISM	Industrial, Scientifical and Medical radio band
LLN	Low-Power and Lossy Network
LLSEC	MAC Link Layer Security
LoWPAN	Low-Power Wireless Personal Area Network
MAC	Medium Access Control

ND	Neighbor Discovery Protocol
ND6	Neighbor Discovery Protocol for IPv6
OSI	Open Systems Interconnection
PAN	Personal Area Network
PLC	Power-Line Communication
PHY	Physical layer
RA	Router Advertisement
RF	Radio Frequency
RS	Router Solicitation
RDC	Radio Duty-Cycle
RPL	IPv6 Routing Protocol for Low-Power and Lossy Networks
SLAAC	Stateless Address Autoconfiguration
SLIP	Serial Line Internet Protocol
WSN	Wireless Sensor Network

Bibliography

[1] Inc Gartner. *Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020*. Dec. 2013. URL: http://www.gartner.com/newsroom/id/2636073 (cit. on pp. iii, iv, 1, 5).

[2] M.Minghetti. *Internet of Things, l'anno della svolta*. Apr. 2015. URL: http://marcominghetti.nova100.ilsole24ore.com/2015/04/14/internet-of-things-lanno-della-svolta/ (cit. on p. 1).

[3] ABI research. *More Than 30 Billion Devices Will Wirelessly Connect to the Internet of Everything in 2020*. May 2013. URL: https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne/ (cit. on p. 1).

[4] ITU. *The Internet of Things*. Nov. 2005. URL: https://www.itu.int/net/wsis/tunis/newsroom/stats/The-Internet-of-Things-2005.pdf (cit. on pp. 5, 8).

[5] Z.D.Boren. *There are officially more mobile devices than people in the world*. Ed. by independent.co.uk. Oct. 2014. URL: http://www.independent.co.uk/life-style/gadgets-and-tech/news/there-are-officially-more-mobile-devices-than-people-in-the-world-9780518.html (cit. on p. 6).

[6] C.Faulkner. *What is NFC? Everything you need to know*. Ed. by techradar.com. Nov. 2015. URL: http://www.techradar.com/news/phone-and-communications/what-is-nfc-and-why-is-it-in-your-phone-948410 (cit. on p. 14).

[7] RF Wireless World. *What is LoRa wireless?* Ed. by rfwireless-world.com. Retrieved April 2016. URL: http://www.rfwireless-world.com/Terminology/LoRa-technology-basics.html (cit. on p. 18).

[8] I.Poole. *LoRa Wireless for M2M and IoT*. Ed. by radio-electronics.com. Retrieved May 2016. URL: http://www.radio-electronics.com/info/wireless/lora/basics-tutorial.php (cit. on p. 18).

[9] LinkLabs. *What Is LoRa?* Feb. 2015. URL: http://www.link-labs.com/what-is-lora/ (cit. on p. 18).

[10] IETF. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. Sept. 2007. URL: https://tools.ietf.org/html/rfc4944 (cit. on pp. 23, 48).

[11] IEEE Computer Society. *Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANS)*. Sept. 2006. URL: https://www.cs.berkeley.edu/~daw/papers/15.4-wise04.pdf (cit. on pp. 29–32).

[12] J.T.Adams. *An Introduction to IEEE STD 802.15.4*. Dec. 2005. URL: https://www.sonoma.edu/users/f/farahman/sonoma/courses/cet543/resources/802_intro_01655947.pdf (cit. on pp. 29, 30).

[13] IETF. *IPv6 Addressing Architecture*. RFC 4291. Feb. 2006. URL: https://tools.ietf.org/html/rfc4291 (cit. on p. 36).

[14] IETF. *Default Address Selection for IPv6*. RFC 3484. Feb. 2003. URL: https://tools.ietf.org/html/rfc3484 (cit. on p. 46).

[15] IETF. *Default Address Selection for IPv6*. RFC 6724. Sept. 2012. URL: https://tools.ietf.org/html/rfc6724 (cit. on p. 46).

[16] IETF. *6LoWPANs: Overview, Assumptions, Problem Statement, and Goals*. RFC 4919. Aug. 2007. URL: https://tools.ietf.org/html/rfc4919 (cit. on p. 48).

[17] A.H.Chowdhury et al. *Route-over vs Mesh-under Routing in 6LoWPAN*. June 2009. URL: http://www.cs.berkeley.edu/~culler/papers/ko11contikirpl.pdf (cit. on p. 52).

[18] IETF. *Routing Architecture in Low-Power and Lossy Networks (LLNs)*. Draft. Mar. 2011. URL: https://tools.ietf.org/html/draft-routing-architecture-iot-00 (cit. on p. 52).

[19] IETF. *Neighbor Discovery for IPv6*. RFC 4861. Sept. 2007. URL: https://tools.ietf.org/html/rfc4861 (cit. on p. 53).

[20] IETF. *IPv6 Stateless Address Autoconfiguration*. RFC 4862. Sept. 2007. URL: https://tools.ietf.org/html/rfc4862 (cit. on p. 59).

[21] IETF. *Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*. RFC 3315. July 2003. URL: https://tools.ietf.org/html/rfc3315 (cit. on p. 59).

[22] IETF. *ND Optimization for 6LoWPANs*. RFC 6775. Nov. 2012. URL: https://tools.ietf.org/html/rfc6775 (cit. on p. 62).

[23] IETF. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. Mar. 2012. URL: http://users.ece.utexas.edu/~perry/work/papers/swa-sen.pdf (cit. on pp. 68, 119).

[24] IEEE ComSoc. *Best Readings in Power Line Communications*. Jan. 2015. URL: http://www.comsoc.org/best-readings/powerline-communications (cit. on p. 76).

[25] P.Anker. *Powerline Communications (PLC)*. 2005. URL: http://www.telecomabc.com/p/plc.html (cit. on p. 76).

[26] I.Berganza, A.Sendin, and J.Arriola. *PRIME: Powerline Intelligent Metering Evolution*. Ed. by cired.net. June 2008. URL: http://www.cired.net/publications/workshop2008/pdfs/SmartGrids2008_0115_paper.pdf (cit. on p. 78).

[27] A.Schwager and L.T.Berger. *MIMO Power Line Communications - Narrow and Broadband Standards, EMC, and Advanced Processing*. Ed. by CRC Press. Feb. 2014 (cit. on p. 79).

[28] D.Ngo. *Home networking explained, part 7: Power line connections*. Ed. by cnet.com. May 2013. URL: http://www.cnet.com/how-to/home-networking-part-7-power-line-connections-explained/ (cit. on p. 80).

[29] B.Bourque. *Arduino vs. Raspberry Pi: Mortal enemies, or best friends?* Ed. by digitaltrends.com. Mar. 2015. URL: www.digitaltrends.com/computing/arduino-vs-raspberry-pi/#:QAr20_HDZZxcEA (cit. on p. 82).

[30] A.Dunkels, B.Gronvall, and T.Voigt. *Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors*. Nov. 2004. URL: http://www.dunkels.com/adam/dunkels04contiki.pdf (cit. on p. 84).

[31] A.Dunkels and O.Schmidt. *Protothread - Lightweight, Stackless Threads in C*. Mar. 2005. URL: dunkels.com/adam/dunkels05protothreads.pdf (cit. on p. 86).

[32] A.Dunkels et al. *Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems*. Nov. 2006. URL: http://dunkels.com/adam/dunkels06protothreads.pdf (cit. on p. 86).

[33] A.Dunkels. *uIP – A Free Small TCP/IP Stack*. Nov. 2001. URL: http://www.dunkels.com/adam/download/uip-doc-0.5.pdf (cit. on p. 87).

[34] M.Kovatsch, S.Duquennoy, and A.Dunkels. *A Low-Power CoAP for Contiki*. Oct. 2011. URL: http://dunkels.com/adam/kovatsch11low-power.pdf (cit. on p. 90).

[35] R.T.Fielding and R.N.Taylor. *Principled Design of the Modern Web Architecture*. Apr. 2002. URL: https://www.ics.uci.edu/~fielding/pubs/webarch_icse2000.pdf (cit. on p. 90).

[36] D.E.Perry and A.Wolf. *Foundations for the study of software architecture*. 1992. URL: http://users.ece.utexas.edu/~perry/work/papers/swa-sen.pdf (cit. on p. 90).

[37] J.Ko et al. *ContikiRPL and TinyRPL: Happy Together*. Apr. 2011. URL: www.cs.berkeley.edu/~culler/papers/ko11contikirpl.pdf (cit. on p. 94).

[38] IETF. *The Minimum Rank with Hysteresis Objective Function*. RFC 6719. Sept. 2012. URL: https://tools.ietf.org/html/rfc6719 (cit. on p. 94).

[39] T.V.Chien, H.N.Chan, and T.N.Huu. *A Comparative Study on Operating System for Wireless Sensor Networks*. 2011. URL: http://tif.bakrie.ac.id/pub/proc/icacsis2011/pdf/301.pdf (cit. on p. 98).

[40] T.Reusing. *Comparison of Operating Systems TinyOS and Contiki*. Aug. 2012. URL: http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2012-08-2/NET-2012-08-2_02.pdf (cit. on p. 98).

[41] D.Carels et al. *Support of multiple sinks via a virtual root for the RPL routing protocol*. June 2014. URL: http://jwcn.eurasipjournals.springeropen.com/articles/10.1186/1687-1499-2014-91 (cit. on p. 119).