POLITECNICO DI TORINO

Facoltà di Ingegneria

Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

# Middleware development for IoT networks


**Relatori:**
Prof. Ing. Marina Mondin
Prof. Ing. Roberto Garello

**Candidato:**
Marco Luciano Forestello

Anno accademico 2017-2018

# Acknowledgements

I would like to express my sincere gratitude to my supervisors in CSULA, Marina Mondin and Fred Daneshgaran for their hospitality and their continuous support even across the ocean. The same goes for Roberto Garello, who was always available to give me a hand since my return to Turin. They offered me a possibility to nourish my passion for IoT.

I owe special thanks to my friends, whether in Italy in Argentina or in the USA, they have always been present during my university days making it an unforgettable experience. Lastly, I would like to thank my family, for their unwavering support through the years, even in the darkest hours and until the last seconds.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Society is continuously evolving. In the last two hundred years, humankind had experienced major waves of innovation. It all began with the industrial revolution and the development of factories, railways, air travel and more. Then, the time of electricity arrived, a time of empowerment of the pre-existing industries that allowed mass production and expansion. More recently, the Digital Revolution shook the way people used to live but it was not until the Internet revolution that the world became connected. Personal computers, data networks, unprecedented access to information and communication consolidated a no-return point. Of course, that could not be the end of it, the current generation is experiencing another metamorphic change, the Industrial Internet. Millions of interconnected tiny devices able to act and sense. This new reality brings together intelligent machines, advanced data analysis and human creativity. Sensors are not precisely a new topic, but a decrease in their cost and the major development in cloud platforms and data storage access have allowed the creation of a new Internet, the Internet of Things. A reality in which these so called things allow machines to be self-aware, predictive, reactive and social.

This phenomenon is getting everywhere, such is the case of Industrial machines which are being equipped with a growing number of electronic sensors that allow them to gather information to operate in entitled new and more efficient ways. Analysing information they perceive from the environment, creating a huge amount of data, easily interpretable for specialists, allowing to save time and pushing forward for innovation and improvement.

The environmental monitoring will be more sensitive, which will help to detect and prevent environmental disasters before they can occur, saving lives and reducing the loose of properties. Home automation, intelligent farming and smart cities are now realities. It can also be applied to medicine and construction. Certainly, the word "disconnected" will fade, giving space to a new kind of normality: smart and connected.

In this context, multiple companies and governments have dedicated time and

money in the development of a new digital software platform for the IoT. A structure able to connect the things to actions; able to retrieve enormous masses of data, to storage and to manage in short time. A system able to provide significant analytics through data mining and machine learning in order to improve actions and decisions. Data that could be consumed by web or mobile applications. The hole thing considering data security and privacy, aspects that are a hot topic in a world in which everything is connected.

## 1.1 The project

This project was started in CSULA(California State University of Los Angeles). The university was interested in developing solutions for their new set of TelosB sensors, both for didactic and experimental purposes. There are infinite possibilities regarding wireless sensor networks applications but one of the most troubling aspects remains the same, the connectivity with other networks. That would enable the possibility of consuming sensor data, that can be located in isolated and remote locations, from everywhere and possibly at any time. Moreover, data is meaningless without a proper analysis and sensors are not powerful enough to perform such tasks.

The objective of this thesis is to develop a stack to connect a WSN to the IoT. More specifically, it aims to transform each sensor in a group of web resources, a part of the Web of Things. The cloud offers a natural solution to IoT, a set of distributed services to process and make available sensor data as well as a storage system capable of gathering big amounts of data. So, the final solution will include a local data server to reach each node and cloud connectivity in order to make such data accessible across the Internet.

The final solution will be a gateway that will enable communication to and from the wireless sensor network. As such, the thesis is focused on connectivity and delivery performance rather than battery saving or security, topics that are partially covered but that are out of the scope for this project.

# Chapter 2

# The Internet of Things

## 2.1 Internet of Things

Kevin Ashton was probably the first man to use the term "Internet of Things" during a presentation in 1999 [1]. He exposed his concerns about how almost all data on the internet was being uploaded by people. According to Kevin, we needed to change the way internet was working by enabling machines to sense and to communicate without any kind of human interaction. That, he believed, would be a new and better way to collect data about things and use it to develop a better world, reducing waste and costs and improving the way we live. Almost a decade later, around 2008, the number of connected devices surpassed the number of connected people and gave birth to the Internet of Things (IoT), a new technology paradigm which promised a global network of machines and devices.

A thing can be a smart watch in a man wrist, a microchip implanted on an animal, a product in a shelf of a supermarket. All those objects, that are present in our environment and which leads our society and our economy, can become a thing in the IoT. In order to accomplish so, they need an unique identifier, to be self aware of its environment and capable of communicating with other objects. We can connect with everyday things and learn about them, we can monitor things, search for things, manage things and even control things and make them act.

## 2.2 IoT Architecture

A paper [2] presented in the IoT World Forum in 2014 proposed a 7-layer architecture illustrated in Fig 2.1. Starting from the edge:

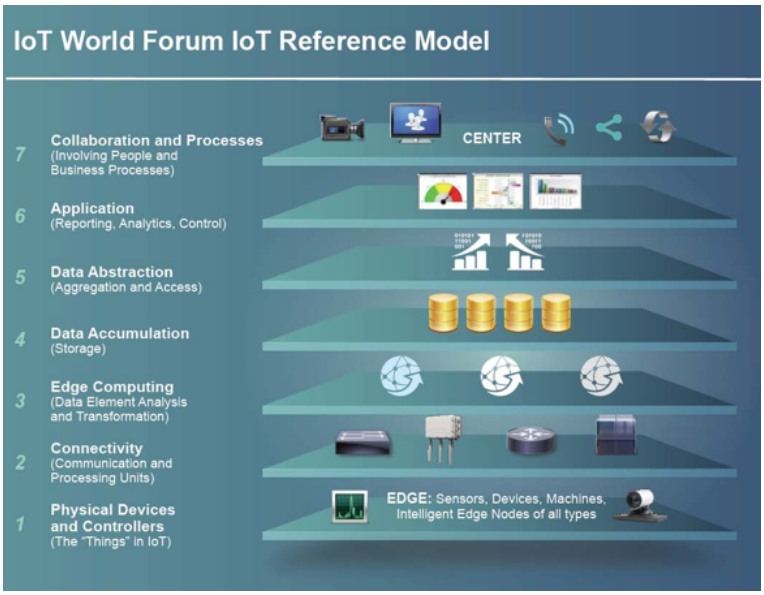



Figure 2.1: IoT World Forum IoT Reference Model

### 2.2.1 Level 1: Devices and Controllers

Physical devices and controllers,that might control multiple devices,are the things of the IoT in charge of sensing and transforming data into digital information. It includes bar codes, RFID tags and reader/writers, sensors, terminals, etc.

### 2.2.2 Level 2: Connectivity

Its task is to transmit the information obtained by the first level in a reliable manner. This includes device-network, across networks and between the network transmissions.

### 2.2.3 Level 3: Edge (Fog) Computing

It is responsible of converting network data flows into information that is appropriate for storage and higher level processing in Level 4. It is not session or transaction aware, it analyses packet units, which limits the processing at this level. Some of its functions are:

- Data filtering, cleanup, aggregation
- Packet content inspection
- Combination of network and data level analytics

- Thresholding
- Event generation

### 2.2.4 Level 4: Data Accumulation

From level 1 to 3 data is "in motion". Data is moving across the network at the rate determined by the devices that generate it. Most applications do not need to process data at network wire speed, they usually assume data is "at rest" in memory or on disk. Level 4 converts data in motion to data at rest. It has to filter useful data for higher levels, determine if such data should be persistent and how it should be saved. Level 4 transforms event-based data to query-based processing, connecting real-time and non-real-time applications.

### 2.2.5 Level 5: Data Abstraction

Level 5 is focused on rendering data and its storage in a simpler way for development, even in a global scale. It has to ensure data consolidation, consistency, authentication and authorization.

### 2.2.6 Level 6: Application

Software in Level 6 interacts with Level 5 and data at rest from Level 4. Application vary according to business needs. Some examples include mobile applications, analytic applications for business decisions, specialized industry solutions and system management/control.

### 2.2.7 Level 7: Collaboration and Processes

All the information created and processed by the IoT is of little value by itself. Applications and its data empowers people to do a better job by analysing and using it. People must be able to communicate and collaborate using the results given by Level 6.

## 2.3 Technologies of the IoT

[3] identifies at least four widely used IoT technologies for the deployment of successful IoT-based products and services: 1) Radio frequency identification (RFID) 2) wireless sensor networks (WSN) 3) middleware 4) cloud computing

### 2.3.1 Radio frequency identification (RFID)

A technology used for automatic identification and data memorization of tags using electro-magnetic fields. It is used to detect presence and location of objects.

### 2.3.2 Wireless Sensor Networks(WSN)

A Wireless Sensor Network (WSN) is made of distributed autonomous devices, also known as motes, capable of sensing and monitoring their environment and communicate captured data to a central or semi-central node over radio waves. Due to its great potential, efficient design and implementation of WSN has become a hot area of research. Recent advances in power saving and wireless communication have allowed to deploy low-cost and low-power networks of wireless sensors, in a vast variety of IoT applications.

### 2.3.3 Middleware

Middleware is a layer situated between software applications.It is typically used in distributed systems to:

- Hide the intricacies of distributed applications
- Hide the heterogeneity of hardware, operating systems and protocols
- Provide uniform and high-level interfaces for applications
- Provide a set of common services that reduce code duplicity and boost collaboration between applications

It is easy to understand how a middleware is fundamental in IoT, considering the widely range of devices, protocols and applications that can be involved.

### 2.3.4 Cloud computing

One of the most important features of IoT is collecting massive amounts of data generated from the things connected to the internet. Furthermore technologies as wireless sensor networks have to handle numerous sensor streams and process data in low-power and low-constraint devices. Cloud computing promotes the delivery of hardware and software resources over the internet using an on-demand utility-based model by meanings of Software as a service (SaaS), Storage as a service (STaaS) and so on. It promises reliability, security, high availability and improved QoS. It becomes the perfect infrastructure of IoT applications that need to store their data on the internet, to perform complex data process subject to QoS constraints or to deliver a service everywhere, at any time.

### Fog-computing

As the number of sensors and the data they generate increase, it becomes more difficult to support all the storage and processing functions of an IoT solution in the cloud. Real-time applications may need faster and more local answers and Edge devices, such as routers, switches and smartphones are able to provide it. The Edge layer is closer to end users than application servers. A new concept of cloud, called Edge Cloud, is enable by processing and using edge devices. The Edge Cloud may help to decrease latency, reduce processing and storage cost and maintain sensible data inside a local context. Fog Computing is an extension of cloud computing that provides processing, storage, and networking services between end devices and traditional Cloud Computing Data Centers, typically located at the edge of network. [4] individuates the key features of Fog Computing as:

- **Edge location, location awareness and low latency** It aims to support applications with low latency requirements (e.g. gaming, streaming).
- **Geographical distribution**: the services and applications targeted by the Fog demand distributed computing and storage resources.
- Very large number of nodes
- Support for mobility to communicate directly with end devices
- Real-time interactions rather than batch processing
- Heterogeneity of nodes and environments
- **Interoperability and federation**: Fog components must be able to interoperate and services must be federated across domains.
- Support for on-line analytics and interplay with the Cloud

2.2 shows a possible integration between Edge layer, Fog computing and Cloud Computing.

## 2.4 IoT applications

IoT promises a network of uniquely identified interconnected devices and not only machine-to-machine but also human-to-machine and human-with-environment interactions. Such a context offers a wide variety of possible applications in different sectors. Gascon and Asin [5] classified 54 different IoT applications under the following categories: smart environment, smart cities, smart metering, smart water, security and emergencies, retail, logistics, industrial control, smart agriculture, smart animal farming, domestic and home automation, and eHealth.

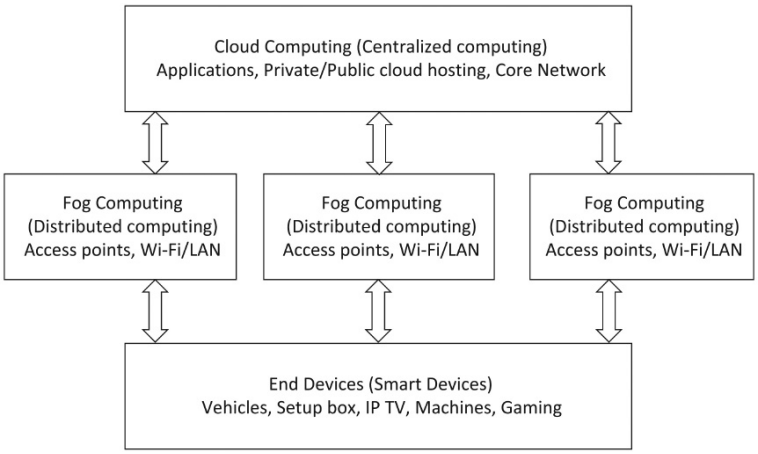



Figure 2.2: Fog computing architecture

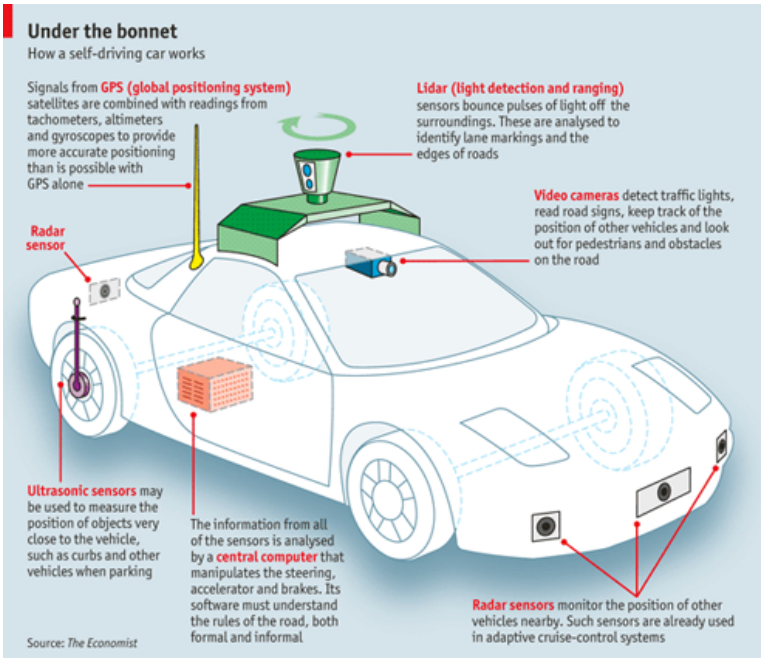



Figure 2.3: Autonomous car sensors

### 2.4.1 Automotive

The autonomous car, one of the most important topics of nowadays, requires an impressive array of interconnected sensors that continuously monitor the vehicle environment and process data locally, in order to take rapid decisions.Fig 2.3 illustrates some of the sensors we could find in a self-driving car.

### 2.4.2 Home automation

Almost every object in a house could be aggregated to a network of interconnected devices under the control of monitoring frameworks that can adjust desired parameters and change configurations based on user's preferences. From security and alarm systems to user movement behaviour, temperature and air monitoring for power saving purposes.

### 2.4.3 HealthCare

Thanks to the enormous amounts of data transmitted from IoT devices to analytics tools it is possible to capture detailed individual health data. IoT enables the personalization of patient care, capturing patients behaviour, hearth rate, blood pressure, glucose level and more. IoT aims not only to collect more data but also to facilitate the way doctors and patients can share it over remote platforms.

### 2.4.4 Sensor Network monitoring

WSNs can be organized over vast areas in order to monitor the environment aiming for greener solutions and trying to prevent or at least rapidly alert cases of natural disasters. Many IoT-based approach systems have been implemented in oceans to control water pollution, providing real-time data over large geographical areas. Similarly, large groups of interconnected sensors monitoring smart cities allow to reduce cost and waste of power, to regulate traffic, to facilitate retailing, etc.

### 2.4.5 Industrial IoT

Also know as Industrial IoT (IIoT) is another form of IoT applications characterized by Machine to machine (M2M) communication, Big Data analysis, and machine learning techniques. These data allow companies to track and manage the supply chain, perform quality control and lower the total energy consumption.

### 2.4.6 Agriculture

IoT enable real-time crops and soil monitoring. It makes easy to adjust the quantity of given fertilizers, water and other products to the soil and the crops according to each sensor network parameters. Thus it reduces impact on nature and plantation cost.

## 2.5 IoT challenges

The IoT is expected to change the Internet and the world but there are many challenges to face before. Some of them are described below.

- **Identification**
  The IoT will connect billions of objects uniquely identified to provide innovative services. Thus, an efficient naming and identity management system is required.

- **Data management**
  The current architecture of the data center is not prepared to deal with the heterogeneity and the amounts of data IoT devices generate.

- **Data mining**
  The development and use of data mining tools become fundamental considering the increase of data provided by the IoT. Besides, more analysts experts are needed to make business decision based on the study of big data.

- **Interoperability and Standardization**
  The IoT counts with a large variety of devices using different sensor protocols, processors, operative systems, programming languages, network protocols and communication protocols. Many manufacturers provide devices using their own technologies and services that may not be accessible by others. The standardization of IoT is very important to provide better interoperability for all objects and sensor devices.

- **Privacy**
  In many IoT applications collects personal data such as users' location, health conditions and personal preferences. One key factor of the IoT is collecting great amounts of data to reduce costs and improve services. At the same time, privacy protection is a main concern to the public. Privacy violation or poor access control to personal data, may lead to a lack of trust in IoT.

- **Data security**
  The attack surface for hackers increase with the number of devices connected to the IoT. A great number of commonly used devices present serious security flaws due to lack of encryption and authorization, insecure Web interfaces and inadequate software protection.

- **Network robustness and security**
  Sensor devices data is sent over wired or wireless transmission network. The transmission system should be able to handle data from large number of sensor

devices without causing any data loss due to network congestion and prevent external interference or monitoring.

- **Spectrum**
  The sensor devices will require dedicated spectrum to transmit data over the wireless medium. Due to limited spectrum availability, an efficient dynamic cognitive spectrum allocation mechanism is required to allow billions of sensors to communicate over the wireless medium.

- **Energy consumption**
  The future IoT will cause significant increase in the network energy consumption. Thus, green technologies need to be adopted to make the network devices as energy efficient as possible.

## 2.6 Summary

As Adam Dunkels, founder of ThingsQuare and Contiki, commented in [6], building applications for the IoT is no easy task. There are a multitude of options at every single step of the way: which kind of device should be used? Which operative system should be installed on it? The same can be said for the network, routing and communications protocols. Another crucial point is the interconnection of devices with the network with SLIP motes, border routers and so on. Even in data processing, Cloud or Fog computing? The options are endless and there is no such thing as a bad or an incorrect solution.

# Chapter 3

# Selected Hardware

## 3.1 The motes

The most popular motes, among the devices used for research and experimentation, are TelosB and MicaZ due to their adaptability to different scenarios, ease of operation and great availability of open source software for them. They are the motes most frequently employed in the implementation of testbeds and are usually the typical platforms used for the validation and assessment of new protocols.

Both motes present a very similar architecture based on a microcontroller, a wireless transceiver and some sensors for measuring physical variables. TelosB includes those sensors on-board while MicaZ allows to add them through expansion connectors and expansion boards.

Both sensors include the same transceiver but they differ on the microcontroller. According to [7] MicaZ motes appeared to be the most performing mote for sending, receiving, and thus relaying operations. In the other hand TelosB has lower power consumption [ 3.1] and lower wakeup time (6 $\mu$s against 4.1ms).

TelosB was chosen for this thesis due to its low power consumption features, its contained price, the high availability of documentation and open source software and for its variety of on-board sensors which allows to deploy multiple services and data on the web.

## 3.2 TelosbB

TelosB is a wireless sensor module developed by the University of California, Berkeley for research and experimentation purposes. It aims for low power consumption, standardization, easy to use and robustness [8]. It is based on a low duty cycle principle that allows the mote to be asleep for the majority of time, until a certain event is triggered. Then, the mote awakes, process and turns to sleep.

| Operation | Telos | Mica2 | MicaZ |
|---|---|---|---|
| Minimum Voltage | 1.8V | 2.7V | 2.7V |
| Mote Standby (RTC on) | 5.1 $\mu$A | 19.0 $\mu$A | 27.0 $\mu$A |
| MCU Idle (DCO on) | 54.5 $\mu$A | 3.2 mA | 3.2 mA |
| MCU Active | 1.8 mA | 8.0 mA | 8.0 mA |
| MCU + Radio RX | 21.8 mA | 15.1 mA | 23.3 mA |
| MCU + Radio TX (0dBm) | 19.5 mA | 25.4 mA | 21.0 mA |
| MCU + Flash Read | 4.1 mA | 9.4 mA | 9.4 mA |
| MCU + Flash Write | 15.1 mA | 21.6 mA | 21.6 mA |
| MCU Wakeup | 6 $\mu$s | 180 $\mu$s | 180 $\mu$s |
| Radio Wakeup | 580 $\mu$s | 1800 $\mu$s | 860 $\mu$s |

Table 3.1: TelosB, Mica2 and MicaZ power consumption [8]

### 3.2.1 TelosB Components

TelosB mote is used for wireless communication. It can measure temperature, relative humidity, and light.

It has a larger chip RAM, which provides robustness in consecution of operation. It is compact in size and has two AA batteries slot, so it can be used anywhere to perform experimentation and research.

It allows external hardware incorporation through its 6-pin and 10-pin expansion connectors.

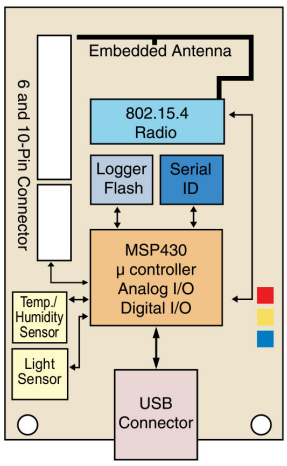



Figure 3.1: TelosB architecture

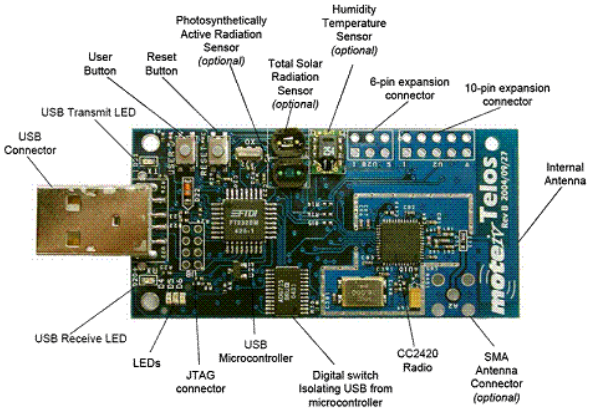



Figure 3.2: TelosB front view

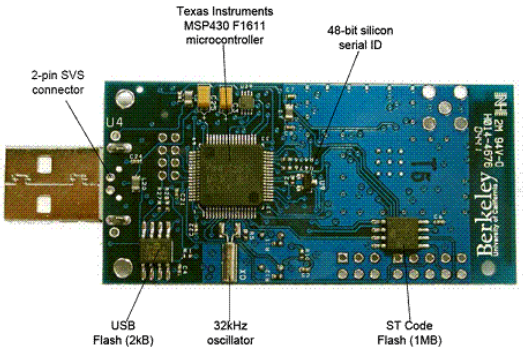



Figure 3.3: TelosB back view

### IEEE 802.15.4 standard radio

TelosB mote has IEEE 802.15.4 standard radio, allowing TelosB to communicate with any other device using the same physical layer. It has a Chipcon CC2420 radio which operates in the 2.4GHz band, with the O-QPSK modulation scheme and with DSSS at 250kbps. It also provides some hardware accelerators to allow encryption and authentication, packet handling support, auto acknowledgments, and address decoding.

### TI MSP430F1611 Microcontroller

TelosB is installed with 8MHz TI MSP430 microcontroller with largest on-chip 10kb RAM. MSP430 has the lowest power consumption, operates down to 1.8V. The microcontroller has the fastest wake-up time, it does not take more than (6 $\mu$ seconds in transition from standby(1 $\mu$ A) to active mode (8 MHz). It also has a DMA controller which reduces the load on the MCU core and an on-chip RAM buffer of 10kB, useful for signal processing.

### 3.2.2 Sensors

TelosB has three sensors:

- Visible Light Sensor: Hamamatsu S1087, Range: 320nm to 730 nm
- Visible to IR sensor: Hamamatsu S1087-01, Range: 320nm to 1100 nm
- Humidity and Temperature Sensor: Sensirion SHT11
  - Humidity sensor range: 0-100% RH
  - Temperature sensor range: -40°C to 123.8 °C

#### Visible Light Sensor and Visible to IR sensor: Hamamatsu S1087, Hamamatsu S1087-01

It is ceramic package photodiodes which give optical measurements i.e. in the visible to near-infrared range. Its measurement can be read in lux, that is the SI unit of luminance, according to the equation:

$$lux = 2.5 * (sensorvalue/4096) * 6250 \tag{3.1}$$

#### Humidity and Temperature Sensor: Sensirion SHT11

SHT11 has two sensors. A capacitive sensor element which measures relative humidity and a sensor which measures temperature. These sensors are coupled to a 14bit A/D converter and a serial interface circuit which gives a fully calibrated digital output.

The relative humidity sensor provides data with a typical 3 ±% accuracy and the temperature sensor has an accuracy of ± 0.5°C.


Figure 3.4: Sensirion SHT11

Figure 3.5: Hamamatsu S1087

The following conversion factors are used to read the sensor values :

$$Temperature = -39.6 + 0.01 * sensordata \tag{3.2}$$

$$Relativehumidity = -4 + (0.0405 * sensordata) + (-0.0000028 * sensordata^2) \tag{3.3}$$

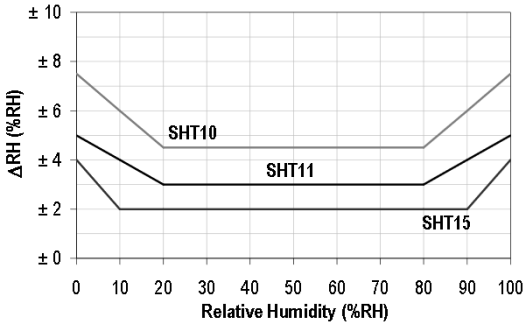



Figure 3.6: Maximal RH-tolerance at 25°C

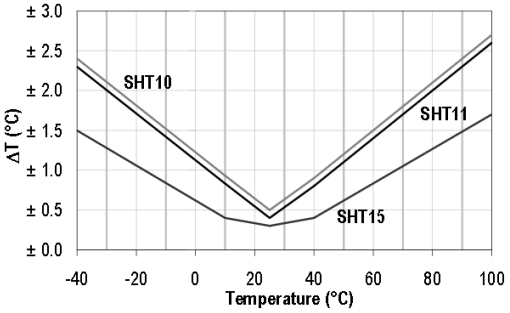



Figure 3.7: Maximal Temperature tolerance

### USB Interface

TelosB can be plugged into USB for programming, decreasing development time, and for power supply. Having on-board USB makes it highly flexible as it can be used on the lab bench, as a gateway to a PC, or as a node in WSNs.

TelosB includes hardware write-protection for external storage which is disabled only when plugged into a USB port. Hardware write protection is essential for systems that may be reprogrammed wirelessly since a known good program image may be stored in the write protected flash.

### Internal connected Antenna

To reduce the cost of expensive external antenna TelosB has internal 2.4GHz Planar Inverted Folded Antenna, directly build into printed circuit board.

### 6-pin and 10-pin expansion connector

Expansion connectors enable users to connect TelosB with additional instruments like LCD displays, digital peripherals, antennas, etc. It offers the possibility to self-configure and boosts the mote for specific tasks or large-scale sensor networks.

## 3.3 Raspberry Pi

The Raspberry Pi is a cheap, flexible, fully customizable and programmable small computer board. It brings the advantages of a PC to the domain of sensor network, what makes it the perfect platform for interfacing with wide variety of external peripherals.

It was created by Eben Upton, Rob Mullins, Jack Lang and Alan Mycroft at University of Cambridge for educational purposes. The Raspberry 3 costs around 35$ and includes [9]

- **1.2GHz 64-bit quad-core ARMv8 CPU**, 50% faster than Raspberry 2
- 802.11n Wireless LAN
- Bluetooth 4.1
- **Bluetooth Low Energy (BLE)**
- **40 GPIO pins**: they are the principal way of connecting with other electronic boards. Some GPIO pins can be used as digital inputs/outputs and as interfaces for embedded protocols such as I2C (Inter Integrated Circuit), a low-speed serial bus interface and SPI (Serial Peripheral Interface) for synchronous full-duplex connections
- Full HDMI port
- Ethernet port
- Combined 3.5mm audio jack and composite video
- Camera interface (CSI)
- Display interface (DSI)
- Micro SD card slot
- VideoCore IV 3D graphics core
- 4 USB 2.0 ports allows connecting peripherals and storage devices
- Micro USB for power

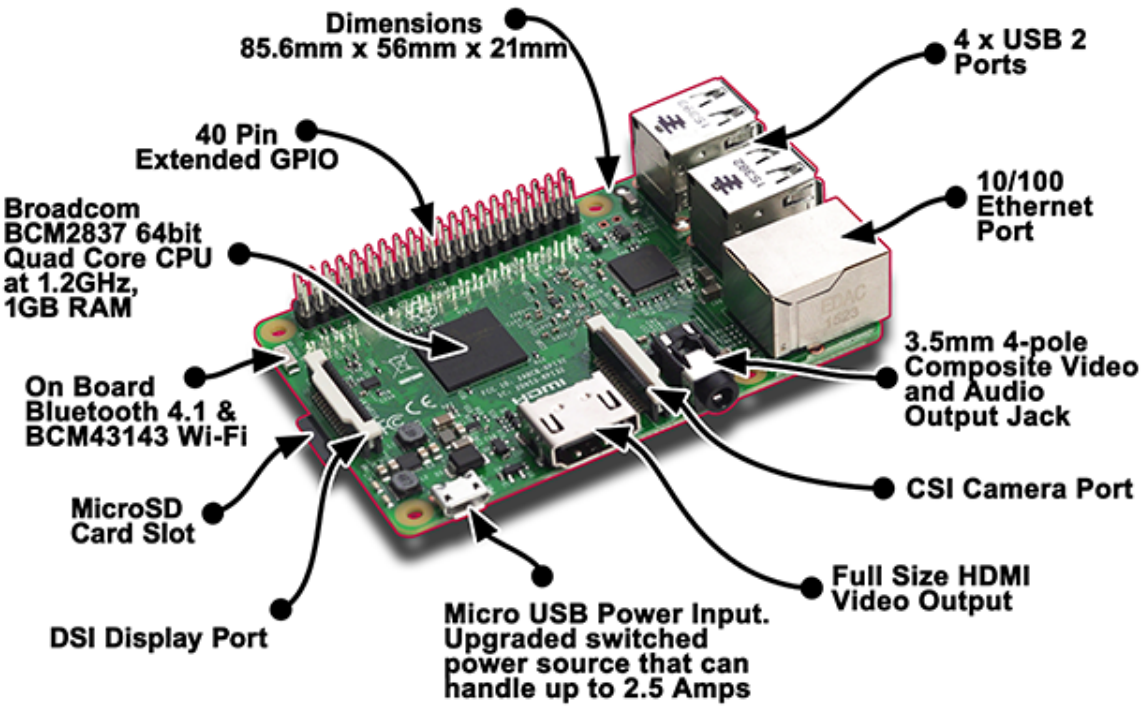



Figure 3.8: Raspberry Pi 3 Model B Specs

### 3.3.1 Communication

A key evaluation metric for any WSN is its communication rate, power consumption, and range. Raspberry possesses multiple ways to connect to a network according to the project needs. It is also compatible with low energy consumption requirements by enabling Low Energy Bluetooth.

### 3.3.2 OS and deployment

Raspberry Pi is extremely easy to use. An OS image and an SD card are all you need to have it running with a Linux distribution,specifically developed for it, called **Raspbian**. Raspbian includes over 35,000 packages and pre-compiled software easily installable on the Raspberry. Raspbian installation is described in subsection A.3.1.

### 3.3.3 Local and remote access

Raspberry Pi can be connected to a screen through the HDMI port, although sometimes is more convenient to access it remotely. There are different ways to do so, the

most common one is through SSH, also known as "secure shell", an encrypted networking technology that enables users to control computers from the command line. It is even possible to remote control the desktop interface of Raspberry, using VNC (Virtual Network Computing), a graphical desktop sharing system. SSH and VNC installation are described in subsection A.3.1. Such flexibility makes Raspberry a good asset for monitoring IoT networks and devices.

In conclusion, Raspberry is a very cheap, very complete and very powerful board able to communicate with other wireless devices, to interact with servers and databases and to act as a webserver, an access point or a gateway/middleware in an IoT application.

# Chapter 4

# Stack Protocol

A main function of every network is to provide some sort of communication among the nodes. When WSNs started to emerge, Internet Protocol (IP) was considered by many inappropriate as it would represent a computational load too high for the sensors, mainly because of header overhead. However, the Internet Engineering Task Force (IETF) created a group in charge of developing an IP based solution on top of IEEE 802.15.4 called 6LoWPAN, acronym for IPv6 over Low power Wireless Personal Area Networks.

### 4.0.1 IEEE 802.15.4

The IEEE 802.15.4 standard defines the physical layer and media access control for Low-Rate Wireless Personal Area Networks (LR-WPANs). It focuses on low complexity, small header overhead and very low power consumption without loosing flexibility. It has been optimized for timing-critical applications with low times regarding access to the network, to the channel and the transient from sleep to active state.

There are two different types of nodes. A full function device (FFD) can be used as a coordinator of a PAN or as a common node in the network. It can forward messages acting as a router. A reduced function device (RFD) may only communicate with a FFD, it has limited memory and less complexity respect a FFD and its limited to star topology only.

At least one FFD must be the coordinator of the network and is in charge of initializing the network, managing the nodes and memorizing information regarding the network topology that can be a star or a mesh.

### Physical Layer (PHY)

The PHY layer provides the data transmission service and an interface to the physical layer management entity. It defines the bands of frequency, the modulation and spreding methods. It operates the RF transceiver, detects a channel availability and executes energy saving and signal power management functions.

IEEE 802.15.4 operates on unlicensed frequency bands:

- **2.4GHz (worldwide)**, data rate : 250 kbps, 16 channels
- **868 MHz (Europe)**, data rate : 20 kbps, 1 channel
- **915 MHz (North America)**, data rate : 40 kbps, 10 channels

### Media Access Control Layer (MAC)

Each device has an unique 64-bit extended address or a local 16-bit reduced address. The MAC layer defines the method of addressing. The MAC relies on Carrier Sense Multiple Access Collision Avoidance(CSMA-CA) as channel access method. It can be a slotted CSMA-CA or an unslotted CSMA-CA, depending on the use of the beaconing signal. Either way, it is possible (and optional) to send an ACK message to confirm data reception. Other functions of the MAC includes frame validation, association to a network and disassociation from it, Granted Time Slots (GTS) management and beacon management.

When the beaconing method is enabled, the coordinator of the network sends periodically beacon signals that the devices must use to synchronize. Between 2 beacon signals a superframe 4.1 is defined, formed by an active part of 16 timeslots and an inactive part. During the CAP (Contention Access Period) devices use slotted CSMA-CA. During CFP (Contention Free Period) devices may ask for GTS (Guaranteed Time Slots).

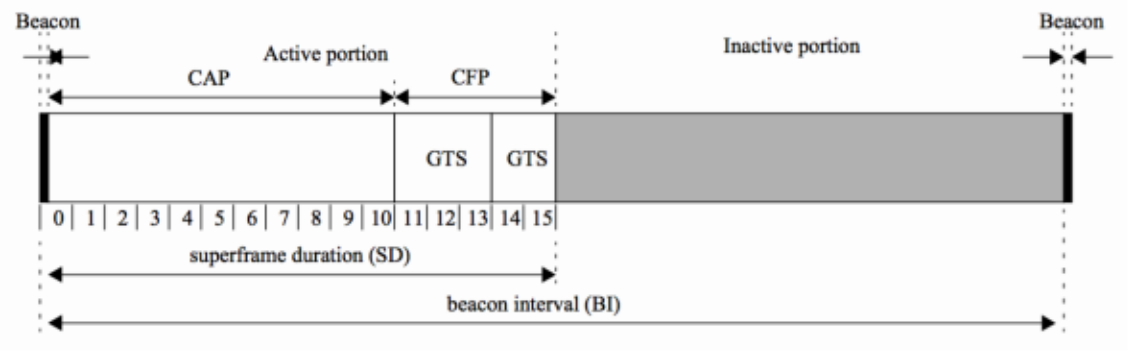



Figure 4.1: IEEE 802.15.4 MAC superframe structure

### 4.0.2 6LowPAN

The 6LoWPAN project was originated on the idea that even the smallest devices should be addressable and thus, able to communicate in the IoT. Unlike the limited address space of IPv4, IPv6, which uses 128-bit addresses, offers $3.4*10^{38}$ unique addresses. More than enough to cover IoT expectations of almost 50 billion connected devices by 2020.

Given the multiple constraints of the motes they cannot apply IPv6 directly. An adaptation layer is needed. 6LoWPAN provides encapsulation and header compression (from 40 bytes of IPv6 to just 2 bytes) in order to allow IPv6 packets to be transmitted over IEEE 802.15.4 radio link.

The 6LoWPAN is connected to the IPv6 network through an edge router which is in charge of handling the data exchange inside and outside the LoWPAN network and the generation and maintenance of the radio network. Since its uses IP, a 6LoWPAN network only needs IP routers to connect to other networks. Other network architectures such as ZigBee or Bluetooth need complex application gateways to communicate to IP networks.

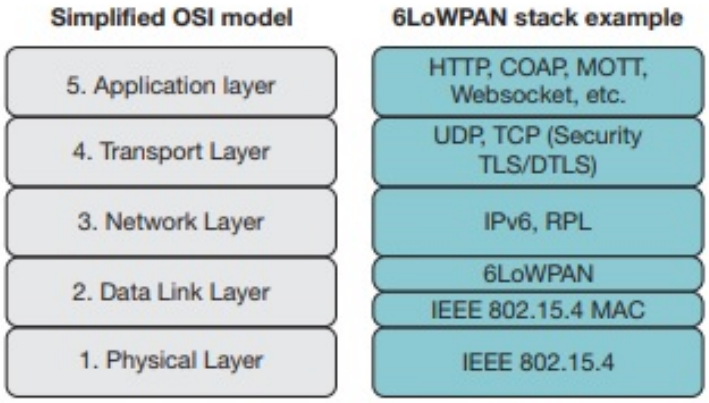



Figure 4.2: 6LoWPAN stack example

#### Headers

6LoWPAN has four different types of headers, individuated by the first 2 bits of the headers:

- **00 - Non-6LoWPAN Frame :** specifies that the following bits should be discarded by a LoWPAN node since they are not LoWPAN encapsulation content.

- **01 - Dispatch Header :** contains a 6 bit selector that indicates the type of the next header :
  - 000001 - uncompressed IPv6 header;
  - 000010 - HC1 header, in other words, a compressed IPv6 header;
  - 111111 - indicates that the following header will be another Dispatch header allowing another 256 types of headers;
- **10 - Mesh Header :** used to send packets in multi-hop networks. It includes a hop limit, source and destination IEEE 802.15.4 addresses. Its length varies between 5 and 17 bytes, depending on the type of addressing used for the communication (short or extended).
- **11 - Fragmentation Header :** in order to allow IPv6 packets to be sent over IEEE 802.15.4, IPv6 frames are divided in smaller segments. Fragmentation headers are generated to reassemble such segments in the correct order. However, fragmentation should be avoided as it has a great impact in devices' battery life.

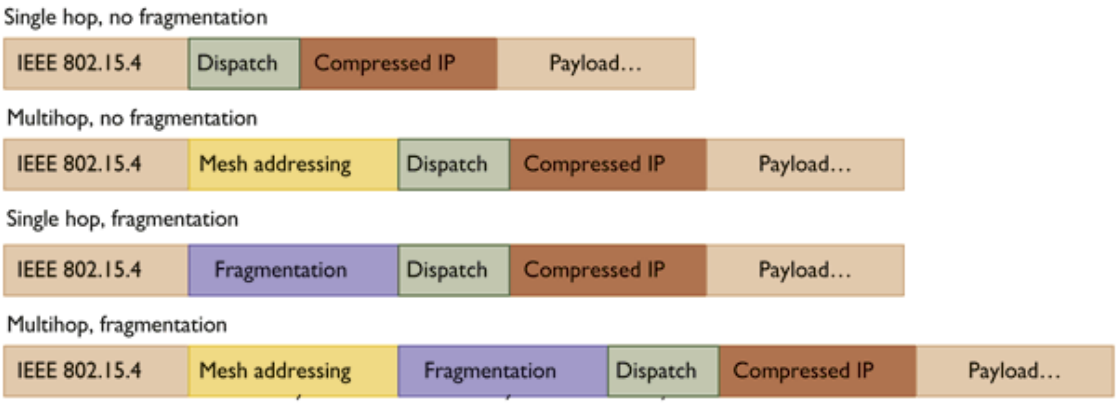



Figure 4.3: 6LoWPAN headers [10]

### Advantages of 6LoWPAN network

An IP based solution is a guarantee of more than 30 years of IP technology development, open standards and applicability of a world wide recognized stack. Furthermore, IPv6 provides a massive address space allowing the creation of large-scale and high-density WSNs. Each node connected to the WSN becomes accessible to any other IP network trough an IP router. Thus, it enables the integration with

standard web service infrastructures such as Representational State Transfer(REST) architecture.

6LoWPAN not only reduce header overhead but also reduce code size and memory requirements. Table 4.1 shows a comparison between 6LoWPAN and one of its major competitors, ZigBee, another protocol built on top of IEEE 802.15.4.

| | Zigbee | 6LoWPAN |
|---|---|---|
| **Code Size with mesh** | 32K to 64K+ | 22K |
| **Code Size w/o mesh** | N/A | 12K |
| **RAM Requirements** | 8K | 4K |
| **Header Overhead** | 8-16 bytes | 2-11 bytes |
| **Network Size** | 65K | $2^{64}$ |
| **RF Radio Support** | 802.15.4 | 802.15.4 |
| **Transport Layer** | None | UDP/TCP |
| **Mesh Network Support** | Zigbee | Many |
| **Internet Connectivity** | Zigbee Gateway | Bridge/Router |

Table 4.1: ZigBee and 6LoWPAN comparison

#### Considerations about 6LoWPAN

6LoWPAN presents some limitations that impact on WSN full integration with the Internet. It only supports IPv6 communication, therefore tunneling and address translation mechanisms should be considered in order to wire the communications to an IPv4 based addressing space. Furthermore, 6LoWPAN does not work properly when firewalls or NATs are involved. Unavailability of static IP addresses, blocked ports and port forwarding may be considered. However, it is possible to handle these issues by using border routers and gateways.

## 4.1 REST architecture

REpresentational State Transfer (REST) architecture defines a set of principles and constraints, design guidelines, to expose web resources. A web resource is an abstraction of every thing or entity that can be addressed, identified by an URI (Universal Resource Identifier). Web resources are managed by the server that allows clients to access and manipulate textual representations of the resources by using a predefined set of stateless operations on the URI.

REST architecture aims for fast performance, reliability, and scalability. Its components can be managed and modified without involving the system as a whole.

REST is usually associated with HTTP. HTTP-based RESTful APIs are defined by a base URL to identify the resourse. Resources are manipulated by means of standard HTTP methods (GET, PUT, POST, DELETE, OPTIONS, etc) and generates representations expressed as internet media types for state transition data elements such as json or XML.

REST allows IoT applications to be developed on top of web services. The sensors become abstract web resources identified by URIs and manipulated by simple HTTP methods. Hence, RESTful WPANs greatly reduce the application development complexity. However, the differences between IoT applications and Internet applications are significant. Low processing power and energy consumption constraints need to be taken into account. As a consequence, the IETF Constrained RESTful environments (CoRE) working group introduced a standardized web service paradigm called CoAP (Constrained Application Protocol).

## 4.2 CoAP

CoAP consists of a subset of HTTP functionalities redesigned to adapt to small devices constraints. It has been built on top of 6LoWPAN and it uses UDP as transport layer. The transport layer is one of the main differences from HTTP, which relies on TCP. TCP's flow control is not suited for WSNs and it has a big overhead for short transactions. UDP instead provides a way lower overhead and supports both multicast and broadcast communications [11].

Table 4.2: CoAP protocol Stack

| Request/Response |
|---|
| Transaction |
| UDP |
| 6LoWPAN |
| IEEE 802.15.4 |

### 4.2.1 Architecture

CoAP is organized in two layers, as shown in Table 4.2. The Transaction Layer handles message exchanges among endpoints. It interfaces with the Transport Layer. The Request/Response Layer handles RESTful request and responses for resources manipulation and transmission. It indicates a Method to access a specific resource and communicates the response Code.

The dual layer approach is the way CoAP provides reliability instead of TCP methods and enables asynchronous communication, a key factor for IoT.

### 4.2.2 Messages

CoAP message format is indicated in Table 4.3. The header is fixed-length (4 bytes) followed by binary options. An usual request has a header of approximately 10-20 bytes which greatly reduces HTTP header overhead.

CoAP establishes an upper limit for the message length. If MTU is not known for an endpoint, it has to be equal to 1280 bytes, the minimum MTU allowed in IPv6.

- *Type* indicates the type of message, Confirmable(0), Non-Confirmable(1), Acknowledgment (2), Reset (3) which are better described below.
- *Code* is a 8 bit unsigned integer that can indicate a request (0), a success response (2xx), a client error response (4xx) or a server error response (5xx). Code 0.00 indicates an empty message, in such a case the packet is only 4 bytes long. Code represent a Request Method for requests and a Response Code for responses.
- *Message ID* is a 16 bit unsigned integer used to detect message duplication and to match ACK/reset messages to the corresponding confirmable or non-confirmable message.

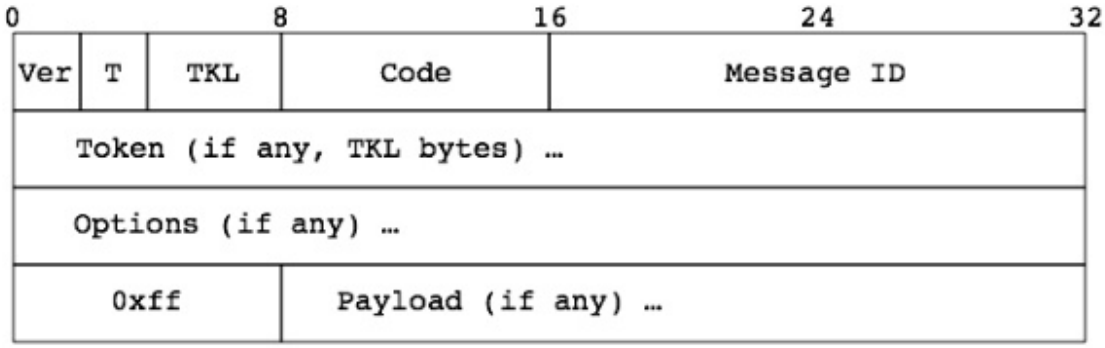



Table 4.3: CoAP message format

Messages exchanged in CoAP can be of four types:

- **Confirmable (CON):** it requires an acknowledgment. It is retransmitted with a default timeout and an exponential back-off until an ACK is received in order not to cause congestion.

- **Non-Confirmable (NON):** it does not require an acknowledgment. It is the case of messages repeated regularly such as sensor readings.
- **Acknowledgment (ACK):** it acknowledges a confirmable message.
- **Reset :** it indicates that a specific message (CON or NON) was received, but some context is missing to be processed. It is a way to test an endpoint liveness, sending a NON empty packet is equivalent of a CoAP Ping.

### 4.2.3 Requests

A CoAP request contains the Method to be used on the resource, the URI of such resource, a payload and Internet Media Type, if needed, and optional metadata.

CoAP handles GET, POST, PUT and DELETE requests, easily associated to those HTTP methods. GET is a safe method so it should only provoke a retrieval action. GET, PUT and DELETE must be idempotent, which means they must generate the same result even if invoked multiple times. POST is not idempotent as it depends on the origin endpoint and the target resource. It usually indicates the creation of a new resource.

### 4.2.4 Responses

After having received and interpreted a CoAP request, a server responds with a CoAP response matched to the request via its token. Such request is identified by a response code which indicates the result of the request operation. There are three types of response code:

- *Success :* the request was received, understood and accepted.
- *Client error :* the request contains bad syntax or cannot be fulfilled.
- *Server Error :* the server failed to fulfill the request.

A response can be sent in different ways:

- **Piggybacked :** the response is sent in the ACK message for the request.
- **Separate :** it is not always possible to return a piggybacked response. This could happen when a server needs more time to process a request or it has to wait for a certain event to occur. When the resource representation is finally available, the server sends the response to the client. If it is necessary not to loose the packet, the server sends a CON message with the response and waits for the client ACK.
- **Non-Confirmable :** a NON response is generated to respond to a NON request.

### 4.2.5 Resource Discovery

A client needs to learn about the server resources. That is accomplished by making a request to a specific CoAP URI. CoAP uses two URI schemes:

- **coap**$-URI$ = "$coap$ : ""//"$host$[" : "$port$]$path$["?"$query$] being the host an IPv6 address or a registered name. Default port is **5683**;
- **coaps**$-URI$ = "$coaps$ : ""//"$host$[" : "$port$]$path$["?"$query$] enables DTLS security. In this case the default port is **5684**;

The path prefix "./well-known/" enables discovery of hosted resources in the endpoint, useful when humans are not in the loop and essential to maximize interoperability.

### 4.2.6 Observe, subscription and notifications

A web resource will likely changes its value over time. CoAP offers a mechanism to follow those changes by "observing" a resource, in other words it allows a client to keep an updated representation of a given resource. A client, the observer, registers at a specific resource which is responsible of keeping a list of registered observers [12].

A client registers to a resource by sending an extended GET request to the server regarding a specific resource. This generates a response with the current representation of the resource and adds the observer to the list of observers of that resource. Whenever the state of the resource changes, after a given timeout or under other events, a Notification is sent by the server with an updated representation of the resource to all the observers in the list.

Fig 4.4[12] shows an example of a CoAP observe exchange. The observe option is setted to 0 the first time and it is incremented every time a notification is generated in order to tell the order of arrival and the representation validity. If a client sends another message with the observe option setted to 1 it means is asking to unsubscribe from the list.

### 4.2.7 Security : DTLS

It is possible to enable DTLS (Datagram Transport Layer Security) over UDP. It supports RSA-AES or ECC-AES.

To initiate a secure communication a provisioning phase is needed. During such phase, the device is provided with the security information needed to establish the connection. Once the provisioning phase ends, the device will be in one of four possible security modes:

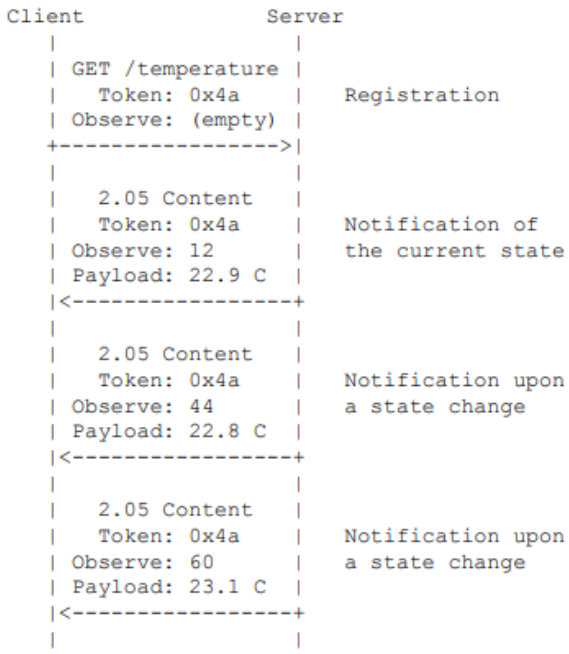



Figure 4.4: CoAP observe example

- **NoSec:** it means DTLS is disabled.
- **PreSharedKey:** DTLS is enabled, each key of a list of preshared keys includes, in turn, another list of nodes in which it can be used to communicate.
- **RawPublicKey:** DTLS is enabled. The device has an asymmetric key pair without a certificate validated using an OOB mechanism.
- **Certificate:** DTLS is enabled and the device has an asymmetric key pair associated with an X.509 certificate.

DTLS is the equivalent of TLS plus some added features to deal with the unreliability derived by the use of UPD as transport layer. In some cases DTLS implies too high of a burden for constraint devices and can not be applied and it does not work for multicast communications.

### 4.2.8 CoAP/HTTP proxying

CoAP supports a subset of HTTP functions. Therefore, it is possible to map CoAP requests/responses into HTTP requests/responses. There are several reasons for proxying between CoAP and HTTP, usually in order to make CoAP resources available for Internet applications. There are two possible directions: CoAP-HTTP

Proxying and HTTP-CoAP Proxying. In both ways, only the model of CoAP is mapped to HTTP.

If the proxy is unable or unwilling to answer a request it returns a 5.05 (Proxying not supported) response to the client.

## 4.3 A Publish/Subscribe protocol : MQTT

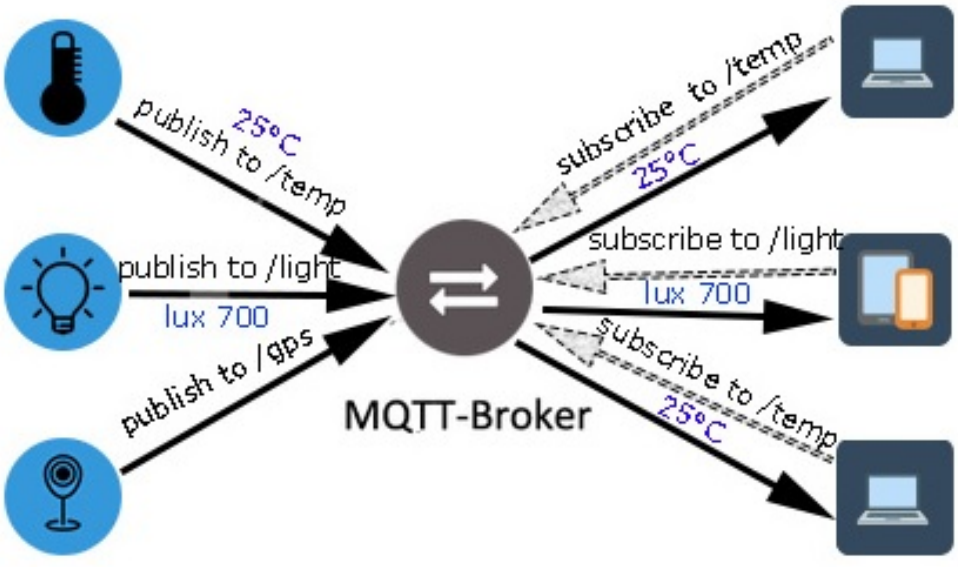



Figure 4.5: MQTT publisher/subscriber model

The publish/subscribe pattern is a simple way of exchanging messages in distributed environments. It is particularly efficient when the connection to the network is intermittent, an event that is frequent in WSNs. It relies on push-based messages and queues to save contents when the connection is not available.

The publish/subscribe design pattern is particularly useful in one to many communications, when a publisher has to update the state of a certain resource to a group of listeners, such as sensor readings.

MQTT, which stands for MQ Telemetry Transport, it is an extremely lightweight (only 2-bytes header) messaging protocol based on the publish/subscribe pattern on top of TCP/IP. It has been specifically designed for Low power and Lossy Networks (LLNs).

The communication is based on a broker (as shown in Fig 4.5). Devices at the edge of a network can publish to a specific topic, in other words an address. Clients can subscribe, or follow, multiple topics. The job of the broker is to filter messages

based on topics and to distribute them to each registered subscriber. The broker allows not only one to many communications but also a de-coupling of applications since there is no direct connection between publishers and subscribers. Publishers just publishes messages without knowing who may be registered to the topic. At the same time, subscribers only express interest in a certain topic without any knowledge about the subscriber behind it.

MQTT is biderectional and it keeps the state of each session. If a publisher loses connectivity, all its subscribers get a notification with the so called "Last Will and Testament" signalling the problem. Then, any authorized client may publish a new value to the disconnected device.

### 4.3.1 Message format

#### Fixed Header

The message header of MQTT messages contains a 2 bytes fixed header which follows the following scheme:

| **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|
| Message Type | | | | DUP flag | QoS level | | RETAIN |
| Remaining Length | | | | | | | |

- **Message Type:** 4-bit unsigned integer
  - *0 - Reserved*
  - *1 - CONNECT:* client request to connect to the server
  - *2 - CONNACK*
  - *3 - PUBLISH:* publish message
  - *4 - PUBACK*
  - *5 - PUBREC:* publish received
  - *6 - PUBREL:* publish release
  - *7 - PUBCOMP:* publish complete
  - *8 - SUBSCRIBE:* client subscribe request
  - *9 - SUBACK*
  - *10 - UNSUBSCRIBE:* client unsubscribe request
  - *11 - UNSUBACK*
  - *12 - PINGREQ:* Ping request

  - *13 - PINGRESP:* Ping response
  - *14 - DISCONNECT:* Client is disconnecting
  - *15 - Reserved*

- **DUP flag:** is set when a client or server tries to re-deliver a PUBLISH, PUBREL, SUBSCRIBE or UNSUBSCRIBE message, in other words, when the QoS value is greater than zero and an ACK is required.
- **QoS flag:** it indicates the level of assurance for delivery of a PUBLISH message. It is further explained in section QoS.
- **RETAIN flag:** is only used on PUBLISH messages. When a client sends a message with RETAIN value equal to one (set) to a server, the latter should hold on the message after it has been forwarded to the subscribers. Upon new subscriptions, the retained message on that topic is sent to the new subscriber, if available, with the flag set. It allows new subscribers to get a last know good value.
- **Remaining Length:** it represents the number of bytes remaining in the current message, the variable header, that may be present between the fixed header and the payload, and the payload.

### Variable Header

Some types of MQTT message includes a variable header that varies according to the message type. The following are some of the fields that may be present in the variable header, MQTT protocol specifications can be consulted for further information [13].

- **Protocol Name:** present in the variable header of a MQTT-CONNECT message.
- **Protocol Version:** 8-bit unsigned integer present in the variable header of a MQTT-CONNECT message.
- **Connect flags:** present in the variable header of a MQTT-CONNECT message, those are
  - *Clean session:* If not set(0), the server must keep subscriptions and in-flight messages for a client after it disconnects. Used for QoS $> 0$ so that the subscribed topics can be delivered when the client reconnects. If set(1), when a client disconnects the server must discard all the available information about that client, including states. A clean session client (flag set) will have to resubscribe each time it connects to the server.

- *Will flag:* it indicates that a message is published by the server on behalf of the client when either an I/O error is found by the server during the communication, or the client fails to publish before the keep alive timer runs out. If it is set(1), Will QoS and Will Retain fields in the Connect flags byte must be present, as well as Will Topic and Will Message in the payload.
- *Keep Alive timer:* present in the variable header of a CONNECT message. It defines the maximum time period between two client messages. Used to detect when the connection to a client is down, improving MQTT reactiviy compared to TCP/IP timeout. If a client does not send any message after one and a half times the keep alive time period, it will be disconnected by the server.
- *Topic name:* it is present in the variable header of a PUBLISH message.

#### Payload

Only CONNECT, SUBSCRIBE and SUBACK MQTT messages posses a payload. In a CONNECT message, the payload contains one or more UTF-8 encoded strings, an identifier for the client, a will topic, username and password (optional) and the message. In the SUBSCRIBE message case, the payload contains a list of topics the client wants to connect to, followed by the QoS levels. SUBACK messages' payload specifies a list of granted QoS for the topic names the server have allowed the client to subscribe to.

### 4.3.2 Topics and topic matching

There is no formal structure for a topic, a publisher has to create topics and can choose both the structure and the topic names. However, there are some naming rules that must be followed, such as [13]:

- a leading '/' creates a distinct topic;
- topic names are case sensitive;
- all topics must be at least one character long;
- the NULL character is not valid in a topic;
- there is no limit to the number of levels in a topic name. However the length is limited to 64k for UTF-8 encoding.

The topic name is present in the header of an MQTT publish message.

#### Wildcards

The structure of the topics is hierarchical. A subscription may contain special characters, wildcards, to allow registration to multiple topics. Multi-level and single-level wildcards can only be used by subscribers.

- **Topic level separator '/':** is used to separate each level of a topic tree providing hierarchy to the topic space structure.
- **Multi-level wildcard '#':** it matches the current topic and all the levels beneath it. It can be specified on its own or next to the topic level separator.

  For example, a subscription to *device1/sensors/#* matches:

  - *device1/sensors*
  - *device1/sensors/light*
  - *device1/sensors/sht11*
  - *device1/sensors/sht11/temperature*
  - *device1/sensors/sht11/humidity* and so on.
- **Single-level wildcard '+':** it matches only one topic level.

  Following the previous example, *device1/sensors/+* matches:

  *device1/sensors/light* and *device1/sensors/sht11*

  but it does not match *device1/sensors/sht11/temperature* and *device1/sensors*.

  As the multi-level wildcard, it must be used next to the topic level separator or by its own. It can also be used within the topic tree (*device1/sensors/+/temperature* is valid).

### 4.3.3 QoS

MQTT offers three different types of QoS levels:

- **0 - At most once:** messages are delivered on best-effort fashion of the TCP/IP network. Lost messages and data duplication are possible. Useful for devices which send messages periodically that can be lost without further implications.
- **1 - At least once:** guaranteed delivery of the message, confirmed by an ACK, although message duplication is possible. If the ACK message is not received or if there is a well known failure in the network or the device, the publisher resends the message with the DUP flag set until it gets an ACK response.

- **2 - Exactly once:** it ensures that the message will arrive only once. It is the highest level of delivery, used when duplicate messages are not acceptable. It generates more network traffic, PUBREC, PUBREL and PUBCOMP messages are needed.

#### Message ID

When the QoS value is greater than zero, when an acknowledge is needed, a Message ID (16-bit unsigned integer) must be present. It must be unique in the set of "in-flight" messages in a particular direction of communication.

#### Message delivery retry

There are some cases in which TCP may fail to deliver MQTT messages. If a response is expected (QoS $> 0$) and it is not received after a certain time period (a configurable option), the sender will try to resend the message with the DUP flag set. If a client looses connectivity and regains it ("on reconnect"), both the client and the server will resend any previous "in-flight" message.

## 4.4 Gateway Framework

"Deploying and configuring one device to act as a node in the Internet of Things is relatively easy. Doing the same for hundreds or thousands of devices is not so easy though" [14]. This is where gateways come in. Advanced software frameworks operating on the edge of an IoT solution, providing developers an abstraction and isolation layer from the complexity of hardware and networking systems, guiding the development and reuse of integrated software and hardware solutions.

### 4.4.1 Gateway tasks

#### Hardware and field abstraction

An IoT gateway must provide easy access to I/O and sensor connectivity.

#### Network management

As a project grows, it is going to need more ways of connectivity and the gateway should provide it with the means to configure and secure it. It also has to consider the actions to perform when the connection is down providing an offline mode.

#### Applications management

One of the main advantages a gateway provides is the ability to remotely manage applications, whether by starting, stopping, installing, updating applications or by controlling configuration parameters.

#### IoT connectivity management

Both CoAP and MQTT are great protocols for IoT but they focus mainly in the communication piece lacking some interactions in offline modes. The gateway should fill those gaps through data buffering, connection retries, provisioning, credentials and certificates.

### 4.4.2 Gateway challenges

[15] individuated three major challenges an IoT Gateway must overcome: fragmentation, complexity and lock-in.

#### Fragmentation

There are multiple solutions, handlers and protocols for every pile of the stack. Starting by the sensor communication protocols we may consider ZigBee, Bluetooth/BLE or 6LoWPAN amongst many others. They are all gaining strength in the IoT world and are good solutions according to the situation. A good gateway can't afford to loose flexibility by preferring a protocol over the other and things can get tricky when trying to interface them all.

On the controller side, multiple operative systems, compilers, programming languages and processors must me considered under the gateway. Communications on the network depends on budget and technologies, they can run over cellular, Wi-Fi, Ethernet, proprietary networks and so on. Multiple standards can be manipulated and there are different choices regarding the communication protocol such as MQTT, CoAP, LWM2M.

#### Complexity

Starting again from the sensor level, it is not an easy task to develop low level APIs to communicate with the sensor OS and HW while respecting its time dependencies. It is necessary to consider cross compiling, the difficulty to debug, test and provide on the control level. Standards usually get driven by customer, varying from project to project. Going up in the stack, other problems such as security, authorization and authentication, reliability and network stability become crucial in the development of an IoT Gateway.

**Lock-in**

Hopefully, as IoT grows there will be less proprietary solutions that limits the development of IoT. But there are multiple cases of proprietary solutions all over the stack. Certain gateways impose the use of proprietary network protocols, communication protocols or cloud frameworks. Cases in which network equipment is locked by carrier and any kind of modification incur high costs or black box devices with or without APIs exposed for the users.

### 4.4.3 Eclipse Kura

Kura is an open source platform developed by Eclipse IoT for building IoT gateways. It allows remote control of the gateway while providing a variety of APIs to access to the underlying hardware, to manage network configuration and cloud communication.

Kura runs over the Java Virtual Machine (JVM). Java was chosen because it is OS and CPU architecture independent, its code can be written once and used anywhere. Java8 and Java9 are committed to IoT development, to improve the way JVM runs on constrained devices.

Kura employs OSGi, a dynamic component system for Java, basically a container which allows to modularize the code in blocks called bundles. Bundles can be reused, controlled and updated independently of the other pieces of code. Any application in Kura is deployed as an OSGi bundle and runs on the container along with other Kura's components.

Kura architecture is shown in Fig 4.6[1].

Kura provides different services:

- *I/O services:* serial, USB, Bluetooth, position (GPS), clock service for synchronization purposes and an API for GPIO/SPI/I2C.
- *Data transport service:* it allows to connect to a MQTT broker.
- *Data service:* a store and forward service, in a priority queue, for data readings picked up by the gateway and published to remote servers. The publishing system is policy-driven, reducing developers concerns about the underlying network layer and the publishing protocol. Kura uses MQTT for data communication and Eclipse Paho as MQTT client.
- *Cloud service:* provides an API to communicate with remote servers. Cloud service allows to manage request/response pattern over MQTT besides typical publish/subscribe interactions. In addition, it provides the possibility to

[1] eclipse.github.io/kura/intro

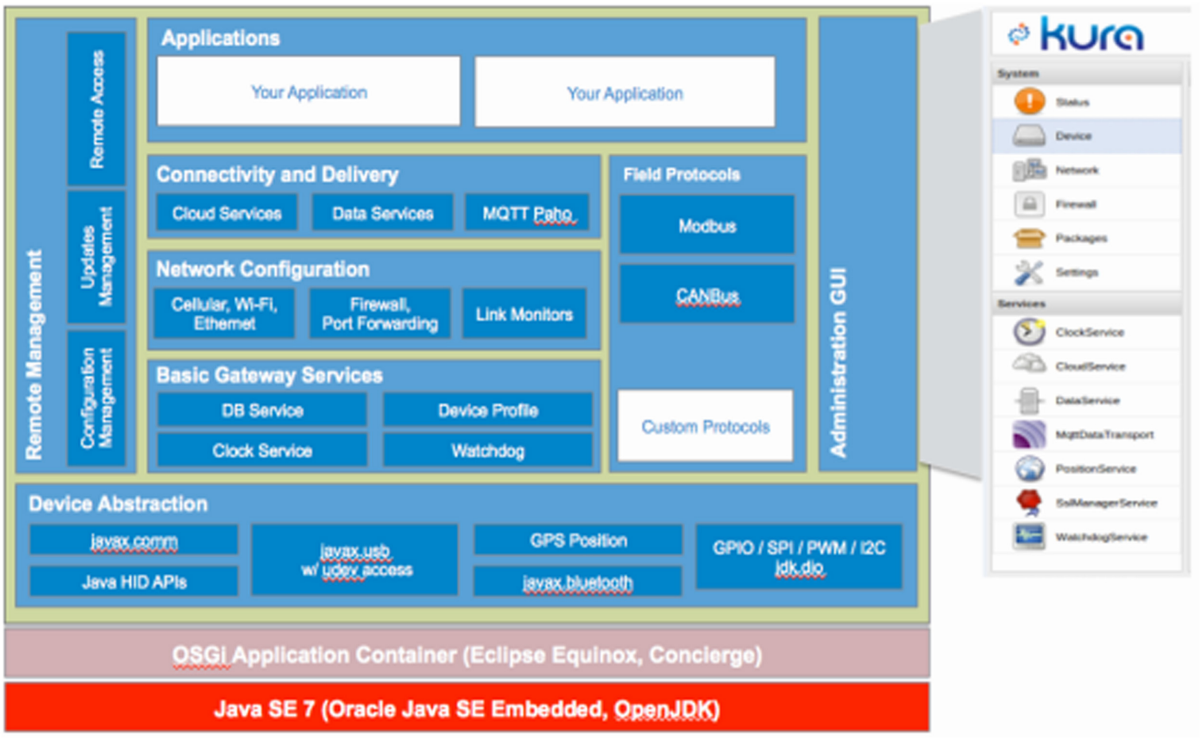



Figure 4.6: Kura architecture

share a connection to a remote server between more applications in the gateway through topic partitioning. It manages message publications life cycle.A data model for the payload is provided by this service which also manage serialization (using Google protobuf), encoding and decoding of the messages.

- *Configuration service:* it provides a snapshot service to import/export the configuration of all the services in the gateway.
- *Remote management:* is it possible to remotely deploy, upgrade, start and stop Kura bundles and to configure system and network properties.
- *Networking:* it is possible to configure all the interfaces available in the gateway (Ethernet, WiFi, etc), the firewall and so on.
- *Web administrator interface:* web console to manage the gateway.

# Chapter 5

# Selected Software

## 5.1 SW for the motes

In order to connect WSNs to the IoT or more specifically, to the WoT, is it necessary to start by choosing the software to be installed in the motes to let them be accessible and communicative with its surroundings. Different choices can be made respect to motes connectivity, but before start talking about network or communication protocols it is important to decide which operative system will be implemented on the motes.

## 5.2 Operative Systems for the motes

To enable constraint devices to run purpose-written applications requires an underlying operative system to manage the resources of the device, focusing on energy, usually provided by a battery that must run for long periods of time, and memory, no more than a few kbytes. This is why the hardware of the motes is developed to reduce power consumption as much as possible, possessing limited resources, the microcontroller used as a CPU is one example. Even if all the components of such devices are developed in order to save energy, they can not be running at full power for too long as the consume would be too high.

In a typical application, a mote is responsible for gathering, processing, and transmitting data gathered by its sensors, it may manipulate such data though its actuators. Furthermore, it has to forward routing messages and participate in distributed algorithms. The operative systems must manage resources and timings in an efficient way since many of them need real-time responses. A mote can only execute a single task at the time, the OS must schedule them, allowing to share CPU resources in order to finish multiple programs in the desired time frame.

The requirements for the OS varies according to the needs of each application.

So, installing a program in a mote usually means to flash the mote with a binary formed by the integration between the application and the operative system, which acts as a library. The OS needs to be flexible to handle different requirements.

In summary, the crucial elements a sensor network operative system must address are *Power consumption*, *Limited resources*, *Concurrency* and *Flexibility* [19]. This section considers two of the best known OS for resource constrained devices available for TeloSB (SkyMote), TinyOS and Contiki. More specifically, how they manage sensor networks main requirements and a brief comparison of the two, justifying the choice of Contiki as the OS for the motes.

### 5.2.1 TinyOS

TinyOS is an open source, flexible OS for sensor networks developed at the University of California in Berkeley. It has very low memory requirements as it fits in approximately 400 bytes. The OS library includes services for data acquisition, sensor drivers and network protocols [19].

#### Architecture

TinyOS has a monolithic architecture and is based on components written in NesC language, a dialect of C. In other words, every application, the OS itself and the scheduler are composed by components that collectively form an application-specific image that can be loaded in the mote. Components are wired and interdependent. Since they are well known at compile time, only used components and procedures are included in the final image and there is no dead code. Each component, and the components connected to it, must be present in order to compile or execute the code. As a consequence, if any component of a program must be replaced or updated it is necessary to rewrite the whole application into the mote.

Components provides for three types of abstractions: *Commands*, *Events* and *Tasks*. The main difference between these types is the way they can be called an by whom. Commands and Events provides inter-component communication, a command may request a reading from a sensor while events would notify the completion of such kind of service at a later time. Tasks act as intra-component concurrency representations. A task do not run immediately, it gets posted by a command or event handlers and it is executed by the scheduler later.

Components expose a set of services, basically a list of commands they can send and events they can handle, by using *interfaces*. An interface can be used or provided by components. Providers implement commands while users implement events. Connections between components are expressed through *configurations*. Configurations specify, for each component, which components use the interfaces it exposes

and which component provides the interfaces it uses. They allow to incorporate more components into a bigger component exposing a single set of interfaces.

#### Scheduling

As previously mentioned, TinyOS provides concurrency by using tasks that can be posted and executed later. The scheduler handles tasks in a non-preemptive FIFO manner. Tasks must run to completion to allow the scheduler to execute another task which allows to use a single stack for all tasks. It is possible to change the scheduling policy. When the queue of tasks is empty the mote enters in a sleep state until it receives an interrupt. It is important to avoid long tasks that may block the whole application since they can not be preempted. At the same time, it is necessary to consider the overhead provoked by posting and executing multiple short tasks.

#### Power consumption

The execution model of TinyOS is event-driven and split-phase. A request for a service is decoupled with the signal of its completion. Therefore, the request returns immediately, as well as the signal event after it notifies the completion of the service. In addition, every time the queue of tasks is empty, the sensor enters in a sleep mode. These two factors help to reduce energy wasting for the CPU. Besides CPU, periphery hardware such as the radio, introduce a considerable waste of energy. TinyOS introduces interfaces to induce a low power state for those components.

### 5.2.2 Contiki

Contiki is a lightweight, portable and open source OS for WSNs developed at the Swedish Institute of Computer Science by Dunkels. It is written in C language, it is based on a driven event kernel and it supports preemptive multithreading. A Contiki configuration occupies approximately 2kB of RAM and 40 kB of ROM.

#### Architecture

Contiki has a modular architecture. It is composed by the kernel, libraries, the program loader and a set of *processes* differentiated in application programs and *services*. Unlike services, application programs do not offer functionalities to other processes. A process must save its state into private memory and the kernel only keeps a pointer to that state. Communication between processes always flows through the kernel. Device drivers and applications communicate directly with the hardware [20].

Contiki allows dynamic loading and replacing of any application program or service at run-time through a dedicated relocating function. Every program has an identification to allow this function to select it and manage it.

A running Contiki system is divided in the OS core and the loaded programs. The kernel, the program loader and the principal libraries and drivers are located in the core, which is typically deployed as a single binary while the programs are distributed independently [20].

### Concurrency and execution model

Contiki follows the event driven model at kernel level although it facilitates thread programming. The execution of a process can be triggered whether by a event sent by the kernel or rather by a polling mechanism, useful to avoid race conditions. The kernel can not preempt events that have been scheduled, but event handlers may use internal mechanisms to preempt other events. Each process must implement an event handler and can optionally implement a poll handler. Polling can be considered as a priority queue for scheduling in between asynchronous events[21].

Both synchronous and asynchronous events are supported. Synchronous events are immediately dispatched to the target process and scheduled while asynchronous events are enqueued by the kernel are dispatched later, in a similar way to tasks in TinyOS.

In addition to events, Contiki offers *Protothreads* for multithreading [22]. Protothreads are a simplified and lightweight version of normal threads. They save their own state in private memory since they do not possess a stack. Protothreads event handlers can not be preempted and run until the protothread goes itself into a waiting state until reschedule [22]. However, Contiki provides a library on top of the kernel in order to support preemptive multithreading. Unlike protothreads, the design used in this library requires every thread to has its own stack and the use of stack switching functions, the timer and interrupt signals. Linking the preemptive library means a greater consume of RAM space respect to the event-driven kernel and code adaptation.

### Communication support

Contiki represents communication support as a service. Like every other service in Contiki, multiple communication stacks can be dynamically loaded or replaced. Furthermore, such stacks can be divided into different services enabling run-time replacing of individual parts of them [20].

Contiki was one of the first sensor network OS to support IP protocols, more specifically IPv6, even if it offers IPv4 and non-IP (RIME) communication.

#### Power consumption

Contiki does not offer any explicit power saving method. The application must decide when it is better to put the mote or some services as the radio into sleep mode.

### 5.2.3 TinyOS and Contiki

A comparison can be made considering the major factors of concern in sensor networks [23]:

- **Power consumption:** TinyOS has implementations for energy conservation, decreasing the burden on the programmer. Contiki instead, does not provide an energy saving mechanism but similar results can be achieved by the use of sleep functions in the application.

- **Limited resources:** Both operative systems can run on constrained devices. However, TinyOS requires fewer space, less complexity and it is better suited when the resources are very limited.

- **Concurrency:** TinyOS offers only the event-driven kernel while Contiki provides different libraries on top of it to allow multithreading.

- **Flexibility:** Both TinyOS and Contiki are flexible to handle different types of programs but, since Contiki was developed for run-time deployments, it excels in applications update and replacement. Its modularity allows to replace fragments of applications without involving the core services.

To bring TinyOS into a mote it is necessary to specify which components to use, how they are wired and to configure each component correctly. Through a NesC compiler it is possible to take all NesC files of a program, including TinyOS OS, and compile them into a single C file. Contiki, on the other hand, requires to write the boot up code, device drivers and parts of the program loader. Since it is written in C, a native C compiler can be used [23].

In conclusion, both operative systems can fulfil WSNs constraints. They are lightweight and flexible. Since both are open source and have been promoting WSN development for years, they count with the support of a big community of developers. TinyOS's community probably is bigger but Contiki has a full set of program examples and tutorials to start with. In addition, they have their own simulation program, TOSSIM for TinyOS and Cooja for Contiki, to simulate WSNs before deployment on the devices.

### 5.2.4 Motivation

Having seen their properties and leaving aside small differences in their performance on really limited devices, they are both valid operative systems for the project. The main difference resides in the possibility that Contiki offers to dynamically link and update services. This thesis aims to be a milestone for a bigger implementation of WSNs in University like structures. It becomes important to allow an easier deployment and replacement of not only application programs but also parts of communication protocols and network protocols. This will facilitate experimental research, evaluating and comparing different solutions. In addition, Contiki provides a protocol for *Over the air programming*[20]. It could allow to remotely replace services in the motes of a sensor network. This implies that sensors, actuators and services can be accessed and modified over the cloud. Moreover, Contiki is written in standard C, development is easy and fast and Instant Contiki provides an entire development environment installation in a virtual machine.

## 5.3 Contiki stack configuration

### 5.3.1 Contiki directory structure

The following is a simple guide of folders of interest in Contiki's directory structure, necessary to better understand the stack and its possible configurations:

- apps/ : Contiki applications
- **core/** : Contiki core code
  - lib/
  - loader/
  - sys/
  - **net/**
    - **mac/**
    - **rime/**
    - **rpl/**
- cpu/ : Contiki CPU-specific code common to all platforms with the same microcontroller
- **platform/** : contains platform specific code
  - **sky/** : contains SkyMote (TelosB) specific code

- doc/ : Contiki documentation
- examples/ : Contiki examples
- tools/ : Contiki tools

### 5.3.2 Contiki MAC layer

Contiki's medium access implementation consists of three layers: Framer, Radio Duty-Cycle (RDC) and Medium Access Control (MAC) as shown below[1]:


Figure 5.1: Contiki MAC layers

All three of them must be defined during compilation time through global variables in *core/net/netstack.h*. All framer, RDC and MAC implementations can be found in *core/net/mac*.

[1] http://anrg.usc.edu/contiki/index.php/MAC_protocols_in_ContikiOS

### Framer Layer

It is a set of functions used to create a frame containing data to be transmitted and to parse received data. There are two types of framers: *framer-nullmac* and *framer-802154*.

When IPv6 is used, framer-802154 is selected. It creates and parses data frames according to the IEEE 802.15.4 standard. It inserts and extracts data from the packetbuf structure.

Framer-nullmac is a very simple framer and it should be combined with `nullmac_driver` of the MAC layer. In this case, the implementation just fills the receiver and the sender addresses of the nullmac header.

### RDC layer

The Radio Duty-Cycle layer manages the sleep periods of time of the motes. It has to ensure motes will be awake to receive messages and it decides when packets must be sent. Contiki RDC implementations can be found in core/net/mac. They are ***contikimac***, *xmac*, *lpp*, ***nullrdc*** and ***sicslowmac*** being contikimac the most commonly used. NullRDC provides a configuration by which the radio is never turned off, therefore there is no power saving and acts only as a pass-through layer. Sicslowmac is another example of a simple RDC layer that do not save energy but it uses IEEE 802.15.4 frames.

Besides nullrdc and sicslowmac, RDC implementations aims to keep the radio off for the longest possible time since it consumes a big amount of energy. Therefore, they check the channel periodically for radio activity. If that is the case, the radio is turned on to control if the packet is addressed to the mote or if it can turn in a sleep mode. The frequency of the medium control, `rdc_channel_check_rate`, is measured in Hz and can be configured.

Controlling the media activity over certain periods of time incurs in packet retransmission or strobe packets since the receiver will not be always awake to receive a packet. As a consequence, the transmitting node will consume more energy and provoke more radio traffic. However, there is an overall power saving for receivers in the network.

### MAC layer

Contiki offers two types of protocols, *nullmac* and *csma*. Nullmac acts as a pass-trough protocol and simply calls the appropriate RDC functions. CSMA (Carrier Sense Medium Access) implements the underlying RDC layer and provides addressing, sequence number and retransmission. It saves a list of packets sent to each mote as well as statistics of collisions, retransmissions, etc. Medium access layers can be

found in core/net/netstack.h in which a global variable for each one of them defines the layer: `NETSTACK_FRAMER`, `NETSTACK_RDC` and `NETSACK_MAC`.

### 5.3.3 Network routing : RPL

There are different implementations of network routing depending on routing metrics or static vs dynamic routing. Although the objective remains the same, to determine the path by which packets will arrive from a source to a destination. The default routing protocol for Contiki is RPL, an IPv6 routing protocol for LLNs developed by the IETF ROLL group. RPL is a proactive distance vector protocol that generates Destination Oriented Directed Acycle Graphs (DODAGs) rooted torwards one sink (DAG root). So data collection is oriented from the source to the sink. A rank is provided to each node in order to determine its distance to the root of the DODAG. The root may acts as a border router for the DODAG, connecting the 6LoWPAN network to other IP networks (as shown in Fig 5.2) . It can aggregate nodes and routes to the graph and redistribute DODAGs routes [26].

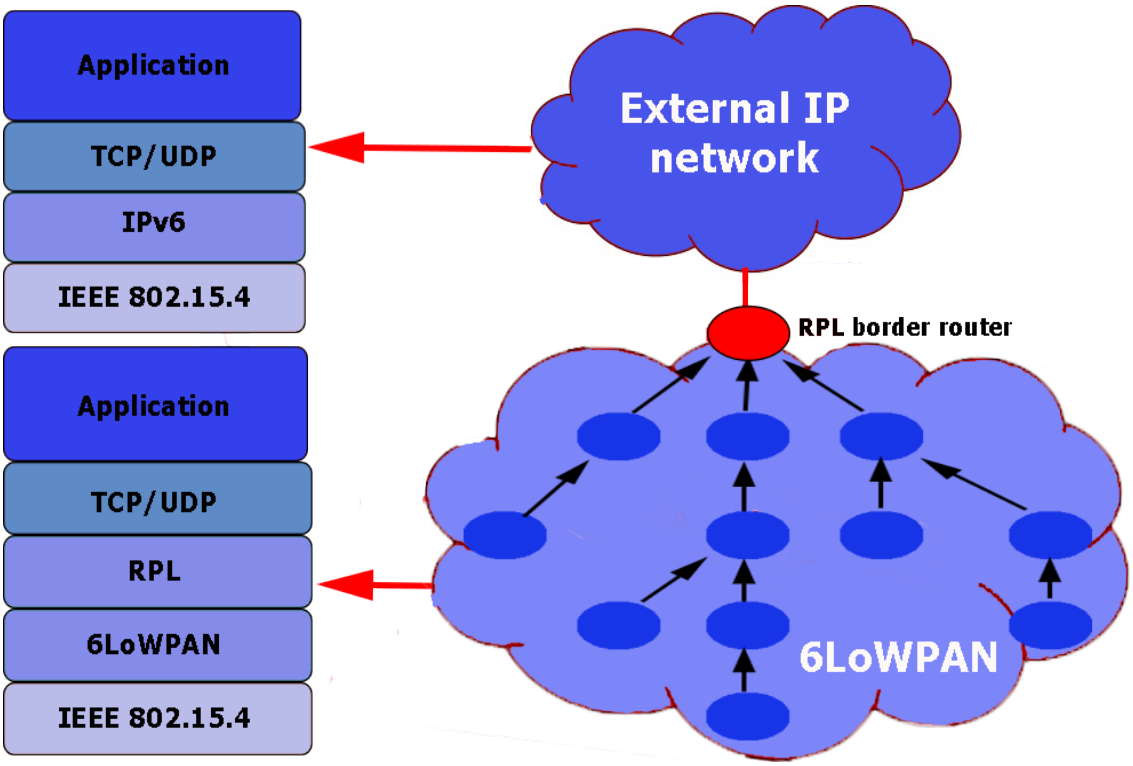



Figure 5.2: An example of a 6LoWPAN connected to another IP network

A network may run multiple instances of RPL. Each one of them is characterized by different parameters in order to considerer multiple constraints or to achieve

better performance. This can be accomplished indicating dynamic constraints and metrics in objective functions. It allows to create optimal routes, according to different criteria, between sink and all the other nodes both to collect and to distribute data.

RPL allows to disseminate network topology information over the DODAG with minimal configuration in the nodes. It sends DODAG Information Objects (DIO) messages in order to build and maintain upward routes of the graph, advertising RPL instance, rank, etc. Nodes must monitor for DIO messages and select a parent node in the network to join the DODAG. Destination Advertisement Objects (DOA) are used to maintain downward routes.

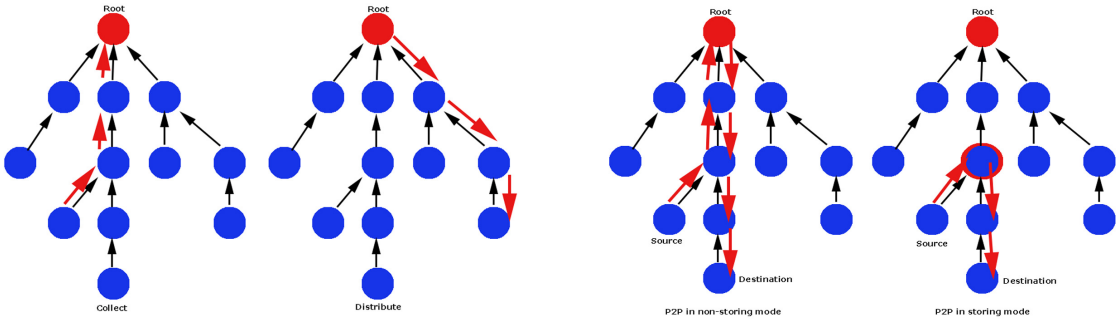



Figure 5.3: RPL: collect, distribute and P2P routing

## 5.4 CoAP implementation: Californium

Californium is an open source Java implementation of CoAP developed by the Eclipse IoT group. It focus on back-end services such as proxies, cloud services or gateways. It provides a set of API for RESTful Web Services that supports all of COAP's features and it is IETF standards compliant. Appendix subsection A.4.3 provides a guide to install and include Californium in the Kura framework.

Californium provides useful features for this project:

- Implementation of CoAP (RFC 7252)
- Implementation of the Observe draft (RFC 7641)
- Implementation of DTLS 1.2 (RFC 6347)
- OSGi wrapper for managed servers

Californium has a three-stage architecture. The stages are decoupled with message queues and are managed with independent concurrency models. Stage 1, the Network stage, depends on OS and transport. Stage 2 handles the CoAP protocol implementation and the third stage manages the logic.

This architecture is implemented by three components: *Resources*, *Endpoints* and *Connectors* [Fig 5.4].

CoAP servers reside in the third stage and contain tree structures in which each node is a resource identified by an address. Each web resource may have a thread pool to handle requests or inherit it from a parent or an ancestor. When a server receives a request, it searches for the resource in the tree. It answers with a response, a response code, according to the search result, and a payload if needed.

Endpoints integrate stage 1 and 2. They enable multiple channels and stack variations for different transports as UDP or DTLS. Endpoints handle datagrams parsing in Requests objects which are forwarded to the resources. The data parser also includes a deduplication matcher to individuate duplicated messages. The CoAP stack implementation of the endpoints includes [24]:

- Blockwise layer: it allows to divide large packets into blocks.
- Observe layer: it handles client-resource relations for the observe feature.
- Token layer: it generates a Token ID for each request.
- Reliability layer: it enables CoAP message retransmission for CON (acknowledged messages) messages.

Connectors reside at the edge of endpoints. They are unaware of the CoAP messages structure as they simply communicates with the transport protocol sending and receiving messages with non-blocking methods.

Californium was chosen as CoAP implementation as it is one of the most complete available implementations. At the same time, it belongs to Eclipse IoT and it is written in Java, which facilitates its incorporation in Kura. Furthermore, Californium provides a full library for security purposes in DTLS called Scandium. This library can be implemented for the CoAP server in the LAN.

## 5.5 MQTT implementation: Paho

MQTT will be used to make wireless sensors data available to the Cloud, and subsequently allow user applications to consume and analyze such data.

Eclipse Paho is the default MQTT client implementation of Eclipse Kura. The Data Transport Service of Kura is based on top of Paho. Paho provides support in many programming languages, Java is one of them.

Paho main features include [25]:

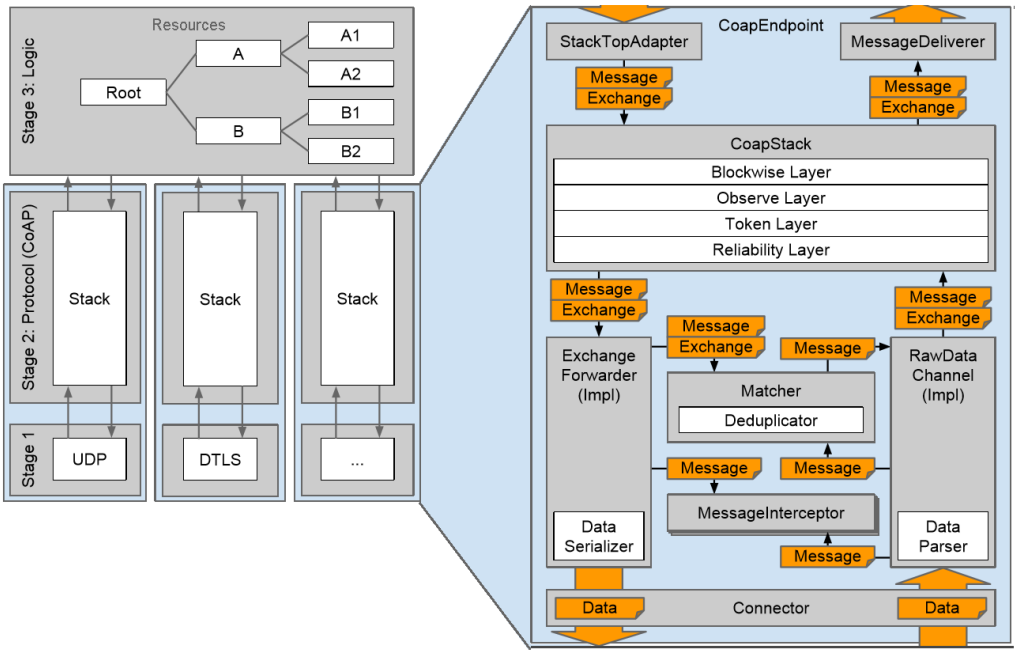



Figure 5.4: Californium Architecture Implementation [24]

- MQTT 3.1
- MQTT 3.1.1
- LWT (Last Will and Testament messages)
- SSL / TLS security
- Message Persistence in case of crash of the system
- Automatic Reconnect to the server
- Offline Buffering
- WebSocket Support
- Standard TCP Support
- Non-Blocking API
- Blocking API
- High Availability

# Chapter 6

# Implementation

In this chapter the implementation of the project is described. The first step is the installation of the CoAP server in the motes followed by the communication to a 6LoWPAN network through the RPL-border router, another TelosB mote which acts as a sink. This mote is connected to the gateway, a Raspberry Pi, which implements the Kura framework. The gateway acts both as a client and as a server. It sends requests to the erbium CoAP servers and it exposes motes web resources to other CoAP clients in the LAN and to the cloud. Fig 6.1 shows the project implementation with AWS as cloud platform.

## 6.1 WSN

This project uses two types of motes, CoAP servers and the RPL-border router. Appendix subsection A.1.1 contains a tutorial to install Contiki programs into the motes.

### Erbium CoAP server

Erbium (Er) is a low power REST Engine for Contiki. Erbium includes an embedded CoAP implementation that became the official one for Contiki. It supports **RFC 7252** together with blockwise transfers and the observe feature [28].

The *er-example-server* was used as the CoAP server program for the motes. It defines a set of resources according to the mote sensors and capabilities. A Contiki RESOURCE in the coap server is defined by an URI, a method, a name and a brief description of its content, query options and the supported MediaType as shown below:

```
RESOURCE(light, METHOD_GET, "sensors/light", "title=\"Photosynthetic and
    solar light (supports JSON)\";rt=\"LightSensor\"");
```

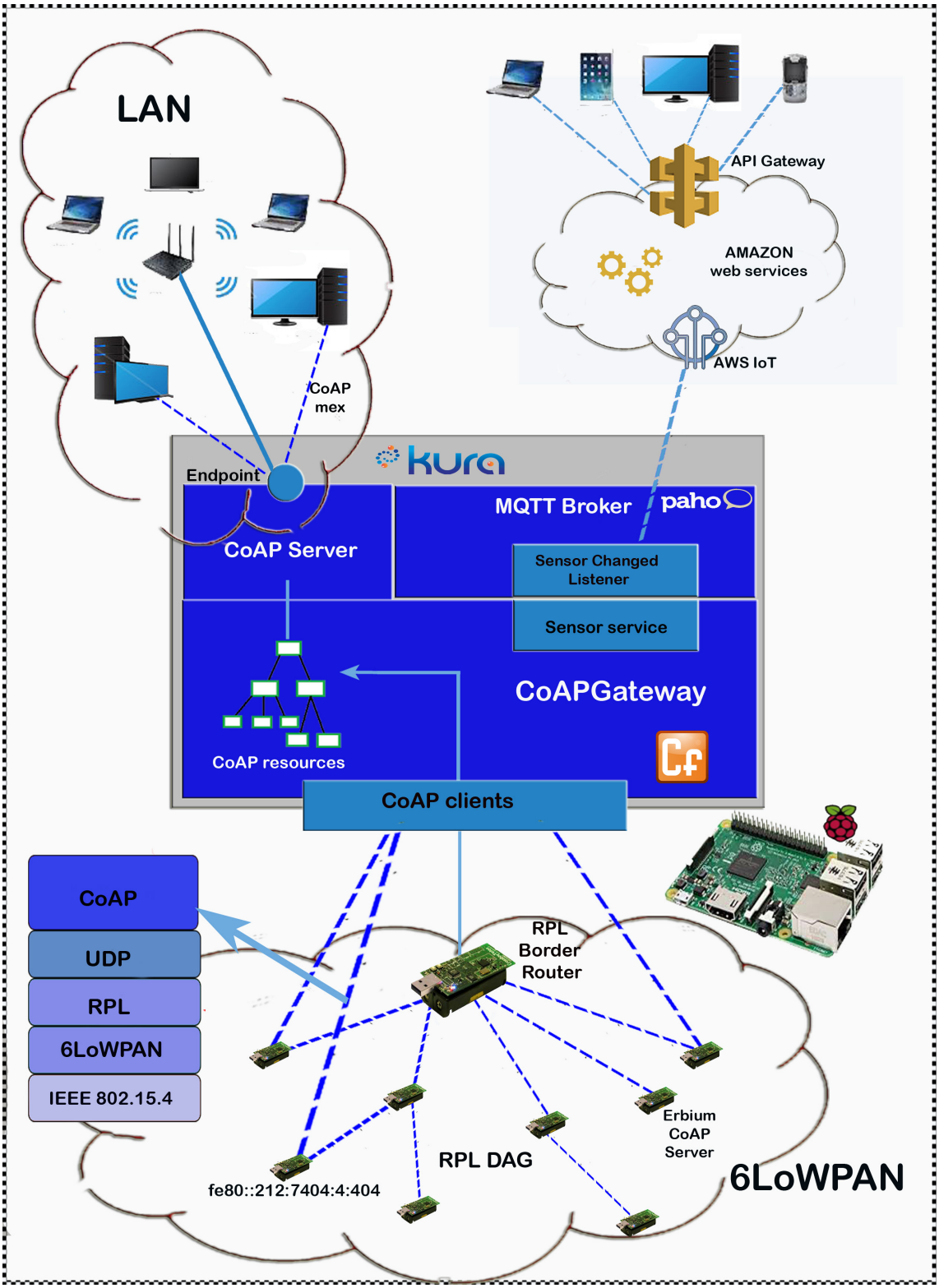



Figure 6.1: Project implementation design

```
RESOURCE(radio, METHOD_GET, "sensors/radio", "title=\"RADIO:
    ?p=lqi|rssi\";rt=\"RadioSensor\"");
```

The server exposes a list of available resources. Some of them can be selected with a flag at the beginning of the file to meet memory constraints:

```
#define REST_RES_HELLO 0
#define REST_RES_CHUNKS 0
#define REST_RES_SEPARATE 0
#define REST_RES_PUSHING 0
#define REST_RES_EVENT 0
#define REST_RES_SUB 0
#define REST_RES_LEDS 0
#define REST_RES_TOGGLE 0
#define REST_RES_LIGHT 1
#define REST_RES_TEMP 0    /* added implementation for the sht11 sensor
    of the TelosB, causes large code size */
#define REST_RES_BATTERY 1
#define REST_RES_RADIO 1
#define REST_RES_MIRROR 0  /* causes largest code size */
```

It is possible to load two or three resources per node in a TelosB mote in average.

When an er-coap server receives a GET request in the "/.well-known/core" it answers with a set of CoRE WebLinks. Each one of them possesses two main attributes: the name and a brief description of the resource containing the allowed methods, query parameters and supported MediaTypes. The resources received by er-CoAP servers are divided in sensors and actuators.

### RPL-border-router

Its function is to communicate the 6LoWPAN network and an external IP network. The *ipv6/rpl-border-router/border-router* was flashed on the border router mote. The program waits until it receives the prefix of the network through a SLIP (Serial Line Internet Protocol)connection with a Contiki tool called tunslip. Once it receives the prefix, it sets itself as the root of a DAG after which it assigns the prefix of the rest of the CoAP servers in the network. Further information about the rpl-border-router setup is available in Appendix section A.2.

The border router hosts a web page with a list of the nodes in the DAG. It contains a list of neighbors, nodes addresses directly connected to the root, and the IPv6 links to reach them and multihop nodes as shown in Fig 6.2.

The border router is connected to the Raspberry Pi via USB, thus the latter can access the RPL network.

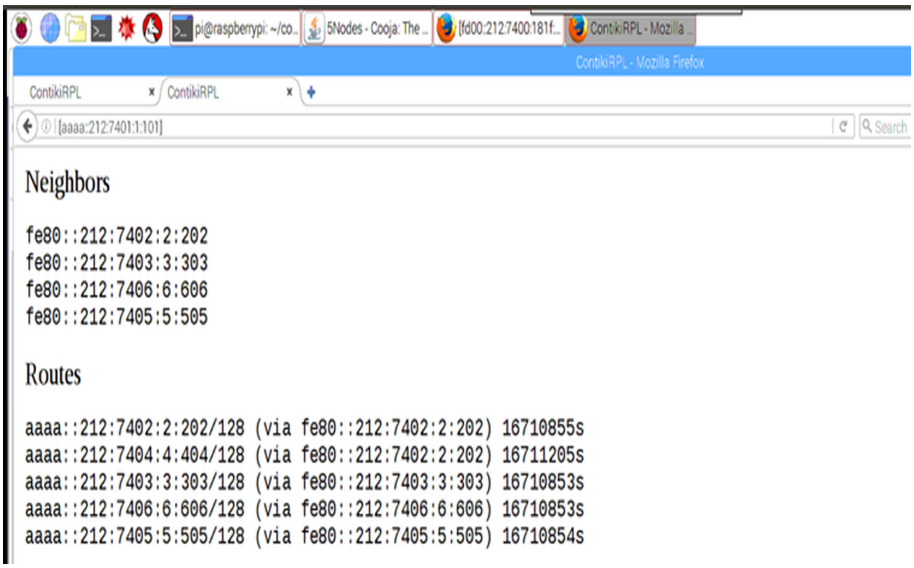



Figure 6.2: RPL border router web page

#### Copper (Cu)

The Copper(Cu) CoAP user agent is an add-on for the Firefox Web browser. It allows browsing, bookmarking, and direct interaction with CoAP resources. by simply inserting a CoAP URI into the address bar [29]. It is a powerful tool to run simulation and interact directly with the connected nodes. Figure 6.2 shows a Copper screenshot with the resources of the CoAPServer offered by the gateway.

### 6.1.1 Cooja simulator

A simulator is a crucial component for the development of wireless sensor network software. The communication or even the debug of physical devices can become tedious and troublesome. In addition, the times for compilation and flash into the mote are high.

Cooja is the default java network simulator for Contiki. It offers a good GUI interface to add different type of nodes in a noise free medium. Different portions of the simulation can be modified without interfering with the process. Cooja offers a packet analyzer and a set of tools to supervise each mote and its communications.

To configure the network as desired it is necessary to add a border router and some servers. The simulation can be linked with tunslip, through a serial server in the border router, in order to start the RPL network and allow communication with the servers.

Once it is active, the CoAP Client in the gateway can starts the communication 6.3. References about the linking between Cooja and tunslip and more configuration

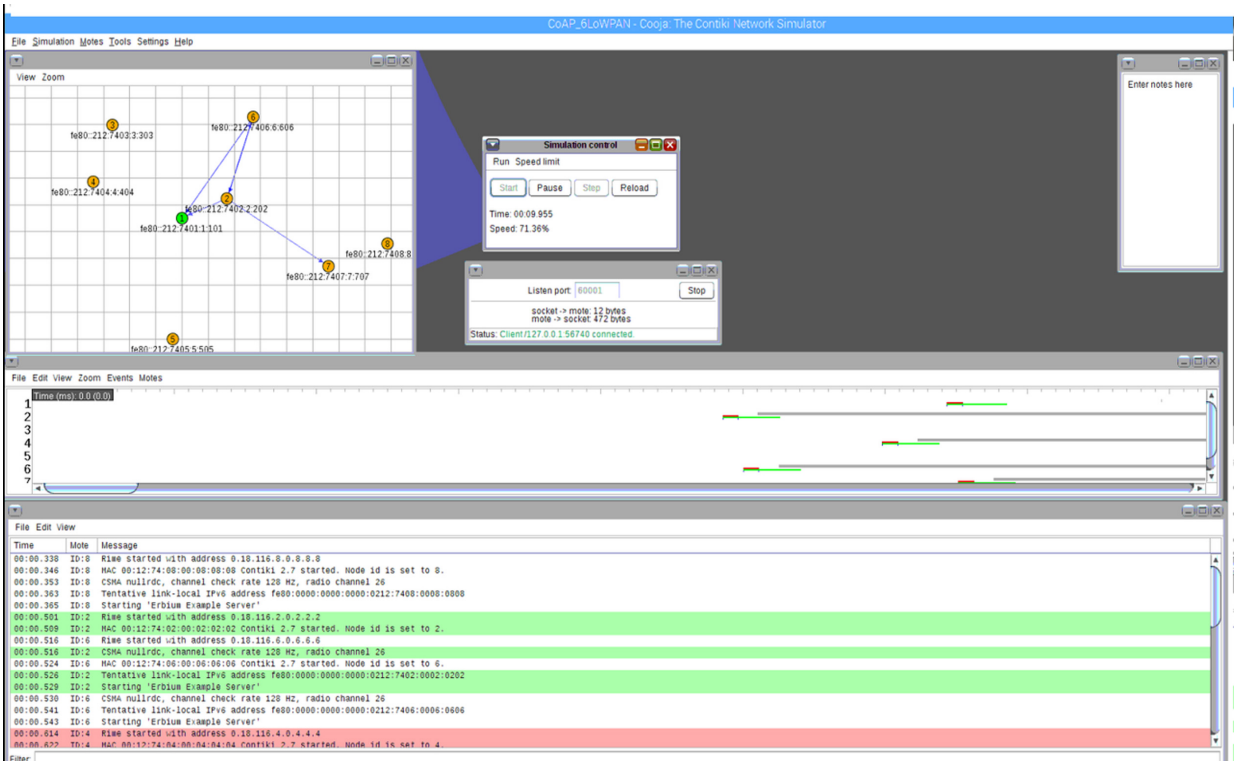



Figure 6.3: Cooja RPL network simulation

details are listed in Appendix subsection A.2.1.

## 6.2 Kura bundles

The implementation of the gateway is defined in three bundles:

- **SensorsInterfaces** : it defines the methods that the other two bundles must implements as a service.

- **GatewayPublisher** : it handles the communication with the cloud sending requests to the CoAPGateway.

- **CoAPGateway** : it is the main bundle. It is responsible for collecting all the data from the sensors. Such data is translated to *CoapResources* that are exposed to the GatewayPublisher and to the LAN through the CoAPServer.

### 6.2.1 SensorInterfaces

SensorInterfaces includes two interfaces to define the behaviour of the CoAPGateway and publisher classes to communicate to each other. This common OSGi approach to allow inter-bundle communication is called Service Registry. Each bundle needs to register a service, in the component XML file, providing its implemented SensorInterfaces interface and bind a reference to the service provided by other bundles.

CoAPGateway implements and provides SensorInterfaces.SensorService and it binds SensorInterfaces.SensorChangedListener. In an analogous mode, GatewayPublisher implements and provides SensorInterfaces.SensorChangedListener and binds SensorInterfaces.SensorService services.

The interfaces define the methods and parameters that allow SensorChangedListeners to request information about the SensorService implementation. Basically, it describes how to retrieve information about the network, its nodes and the resources offered by each one of them. Specific methods will be described in the following sections.

### 6.2.2 GatewayPublisher

It manages the communication between the CoAPGateway and the cloud platform. In order to accomplish such task, it must implement:

- **CloudClientListener**: it is an implementation of Kura Cloud service. It allows to communicate with instances of diferent CloudService Platforms. Cloud connections must be created and configured from Kura web UI, under CloudService. An example of cloud connection with AWS IoT cloud platform will be described in the following chapters.
- **ConfigurableComponent**: it allows to expose a Kura service to modify the configurable properties of the publisher, such as:
  - *publish.qos*: default QoS to publish the messages with.
  - *publish.retain*: default retaing flag for the published messages.
  - *publish.appTopicPrefix*: prefix of topic to publish CoAPGateway responses.
  - *publish.rate*: default publish rate time for observers [ms].
  - *publish.validity*: default publish validity time for resources values [ms].
  - *subscribe.appTopicPrefix*: prefix of topic to receive cloud platform requests.
- **SensorChangedListener**: it implements the methods needed to communicate with the CoAPGateway. Through this interface it can gather and publish

information about the wireless sensor network, its nodes and each resource exposed by them.

As soon as the bundle becomes active and the CloudService becomes connected it subscribes to the inbound/ topic in order to listen requests from the cloud. Cloud application requests are forwarded to the SensorService if active, otherwise it replies with an error message. When a SensorService has to communicate with the cloud platform it sends the message through GatewayPublisher.

When the SensorService is set, GatewayPublisher sends a request to retrieve information about the MoteNodes in the network. Afterwards, it sends a resource discovery request for each mote. All the responses are immediately published by the cloud client to the respective topic. If the connection with the cloud is down all incoming responses from the SensorService are just ignored.

### QoS

It is possible to select the three MQTT QoS leveles for the messages between the gateway and the cloud. Messages published by the cloud client (ClientService) are buffered by Kura DataService until the connection is established. The behaviour is different according to the QoS value:

- **QoS = 0: at most once.** The DataService sends the message only once to the MQTT library avoiding duplicates.
- **QoS = 1: at least once.** In-flight (unconfirmed) messages can be re-published on re-connect but it may cause duplicates.
- **QoS = 2: exactly once.** In-flight messages are deleted when the connection is down so they are not re-published on re-connect. If the connection falls down, the application must assume the message was not received by the broker.

This parameter can be modified when publishing responses from the motes exposed by the gateway. Observed resources do not explicitly need to get their values exactly one time or ensure delivery since they send multiple and frequent values to the cloud. In the other hand, information regarding the 6LoWPAN network must ensure delivery to keep cloud services updated about the CoAP servers that are currently available through the gateway.

### Topic structure

The root of the topic structure used to communicate with the gateway is defined by **"<clientd-id>/appTopixPrefix"**. Client-id is an unique identifier within AWS domain used to the connection to AWS cloud services described in the next chapter.

AppTopixPrefix can be configured through Kura web UI as previously mentioned. The other topics are added to the root prefix creating a tree structure as described below:

- **neighborhood/**: by subscribing to this topic, the connected cloud platform will get periodic updates of the motes exposed by the server. Their address, last seen time, parent node and status.
- **neighbors/<nodeId>**: this topic receives the WebLinks of each mote, which describes the allowed methods and parameters to interact with each of its resources.
- **neighbors/<nodeId>/sensors/<sensorName>**: it receives the reading values of a specific sensor.
- **log/**: it logs CoAP message responses indicating the result of each request.
- **inbound/**: cloud applications must subscribe to this topic in order to make a request.

### Message payload structure

KuraPayload is the standard and recommended payload structure for the messages sent to the cloud. The format was designed by Eurotech as a flexible and efficient open format. The format can be serialized into XML, JSON or Google ProtoBuf.

KuraPayload contains the following fields [30]:

- *sentOn timestamp:* it indicates the timestamp when the data was sent to the cloud platform.
- *metrics:* a set of metric data structures represented as name-value pairs. Allowed types of metric are: string, double, int, float, long, boolean, base64Binary. Sensors responses are represented as metrics.
- *position:* an optional field to capture a GPS position associated to the payload.
- *body:* it is an optional section of the payload that allows additional information to be transmitted in any format defined by the user.

The following snippet shows examples of cloud applications json requests to the motes, published in the inbound/ topic.

```
GET Request to the /radio?p=rssi resource of the mote
    aaaa::212:7400:1539:e48b
{
   "metrics":{
         "method": "GET",
         "uri" : "aaaa::212:7400:1539:e48b/sensors/radio",
         "query" : "p=rssi"
   }
}

PUT Request to the /leds resource of the mote aaaa::212:7400:1539:e48b
{
  "metrics":{
    "payload": "mode=on",
    "method": "PUT",
    "uri" : "aaaa::212:7400:1539:e48b/actuators/leds",
    "query" : "color=r"
  }
}

OBSERVE request to the /light resource of the mote
    aaaa::212:7400:1820:23f2 with a particular fixed rate
{
"metrics":{
    "method": "GET",
    "uri" : "aaaa::212:7400:1820:23f2/sensors/light",
    "publishRate" : 1000,
    "observe" : 0
  }
}
```

### 6.2.3 CoAPGateway

The objective of this bundle is to act as a gateway for the 6LoWPAN network beneath it. It gathers the data from the Telosb sensors and it expose the web resources to both CoAP and MQTT clients. It implements the SensorService of the SensorInterfaces and the ConfigurableInterface in order to expose a configurable service on the Kura web UI.

The main tasks of CoAPGateway are listed below:

- **RPL-DAG control:** it has to initialize the tree of sensor nodes in the network and it runs a worker periodically to check for topology changes.

- **Resource discovery:** after the initialization of the tree of neighbors, it sends a discover request to each one of them in order to know which web resources do they offer. With such information it creates the sensors and actuators resources for each neighbor with their corresponding attributes.
- **CoAP server:** as soon as the initialization of the CoAPResources is finished, it starts a CoAP server bound to the localhost endpoint. Thanks to Kura it is possible to wire the addres to the LAN in which the Raspberry is connected to, enabling CoAP clients to access the server.
- **Gateway Service:** it exposes the same resources to MQTT listeners services, allowing cloud connections and interactions with the motes.

### Data structures

The main data structures can be divided according to its functions in: resources data structures and concurrency control data structures.

Resources data structures are based in Classes that extend Californium *CoAPResource*. CoapResource is a tree representation of web resources. Each CoAPResource contains its children in a ConcurrentHashMap<String, Resource>, a hash table supporting concurrency of retrievals and updates ensuring thread-safe operations[31]. It also possesses a List of observers and yet another ConcurrentHashMap<String, AttributeValues> for the attributes of each resource. Typical attributes include: TITLE, `RESOURCE_TYPE`, `CONTENT_TYPE`, OBSERVABLE, `INTERFACE_DESCRIPTION`.

The data structures used for web resources are:

- **CoapResource neighborsResource:**it contains the tree of all the resources of the 6LoWPAN network. The root is the rpl-border-router which has one *MoteNode* child for each neighbor. Each MoteNode possesses two children: a CoAPResouce for the sensors and another for the actuators. All the resources of each mote are differentiated as *SensorResources* or *ActuatorResources*. Both of them extend CoAPResource. The name of each node reflects the corresponding section of the url for that resource in the tree. Each extension possess a set of attributes for statistical purposes.
  - *MoteNode:* it contains the main information about each mote in the sensor network such as a long indicating the last time it has been seen, its status and its type. Its name is the IPv6 address of the mote. It has two children, '/sensors' and '/actuators'.
  - *SensorResource:* extension dedicated for each sensor of the mote. It saves the sensor value and a CopyOnWriteArrayList<SensorChangedListener> which indicates all the listeners interested in that resource for observation.

It manages server requests to GET the sensor value making a client call to the mote if its value validity time is expired. It handles both synchronous and asynchronous client calls.

- *ActuatorResource:* extension dedicated for each actuator of the mote. It forwards PUT and POST requests from both the global server and the GatewayPublisher to the actual motes.

- **PriorityBlockingQueue <CoapResource> discoverResourcesQueue:** it keeps a priority queue of resources that must be discovered, whether because they just join the tree or for periodically checks for changes within each mote. The former resources have a higher priority than the latter, considering it is more important to know the WebLinks of a new mote. PriorityBlockingQueue guarantees thread safe operations since it defines blocking versions of put and take java:concurrency.

- **Node<String> rplRoot:** it keeps a tree representation of the neighbors of the network as they are reached by the border router.

- **CoapServer coapServer:** it contains *neighborsResource* with all the web resources that exposes to CoAP client in the network.

In the other hand, the structures used to handle the concurrency for those resources and the medium access are:

- **ScheduledExecutorService:** it can schedule commands to run after a given delay, or to execute periodically. By scheduling commands it creates tasks with different delays and it returns a task handler to cancel or check execution [32]. This project uses two different types of ScheduledExecutorService for its concurrency structures:

  - *ScheduledExecutorService observe*:this ExecutorService creates a thread pool that reuses a fixed number of threads, a *Executors.newFixedThreadPool (int nThreads)*. It guarantees that at most nThreads will be active at any given time. Additional tasks submitted when all threads are active will wait in an unbounded queue until a thread becomes available. The threads in the pool will cease to exist only after an explicit shutdown() [32]. *observe* allows to keep a maximum number of concurrent active threads, which results vital considering Raspberry Pi limitations. It executes periodical **observe** requests for each resource, if asked to by the server or the GatewayPublisher.
  - *ScheduledExecutorService borderRouterThreadScheduler :* it is a *Executors.newSingleThreadScheduledExecutor()*. This type of scheduler creates

a single-threaded ExecutorService. Tasks are guaranteed to execute sequentially, only one task will be active at any given time[32]. The *borderRouterThreadScheduler* handles the discovery *discoverResourcesQueue* to periodically perform discover operations updating or creating the CoAPResources for each node.

- **ArrayList<ThreadRequestHandler> threads :**n *ThreadRequestHandler* are running in the CoAPGateway, with n equals to the configurable parameter of the `POOL_SIZE`. Each *ThreadRequestHandler* handles all the requests, both normal (GET, PUT, POST) and OBSERVE, for a certain number of motes. This number depends on the total motes of the network. If such number is lesser or equal than the texttttPOOL_SIZE, each mote will have a dedicated thread. If, in the other hand, the number of motes is greater than the `POOL_SIZE`, the remaining motes will be divided according to the network tree topology controlling *rplRoot*. Parent and children motes will be handled by the same thread in order to avoid mixing the timings of communications of multiple threads on that portion of the network. If their is not parental relationship between motes they are assigned to different threads in order of arrival. Since each thread may handle multiple requests for different motes, a **Semaphore mediumAccess** is implemented to control how many motes can receive requests at the same time.

A last data structure, called **listeners** is used to keep a list of classes that implement the *SensorChangedListener* interface and are connected to the CoAPGateway.

### RPL-DAG control

It is the first operation of the CoAPGateway as soon as the bundle becomes active. It retrieves the address of the rpl-border router and it performs an HTTP GET call to get the addresses of the motes. If the border router does not respond after a given time, the CoAPGateway reloads it and retry until it becomes active.

The CoAPGateway retrieves the list of motes in the 6LoWPAN network and creates the first layer of childs of *neighborsResource* containing an entry for each mote. Later on, it enqueues each *MoteNode* in the *discoverResourceQueue* in order to enable resource discovery.

Lastly, this function controls the status of each node, the insertion of new nodes in the network and the removal. Each time a new node is detected it enqueues the newly created resource for resource discovery. In the other hand, every time a node becomes inactive, it handles the cancellation of the resources and notifies all the interested listeners.

This process is executed periodically in a single thread created through *borderRouterThreadScheduler*. Every time it finishes the topology control, it sends an

updated version of the neighborhood to the GatewayPublisher to keep MQTT listeners informed about the status of each mote. After which, it starts the resource discovery functionality.

### Resource discovery

The border router provides the list of neighbors and their IPv6 links. Knowing each mote server address it is possible to perform a CoRE discover call, in other words, a GET to "/.well-known/core" on the CoAP server. The server returns a payload containing its resources in the CoRE Link Format. These links contains the name, the supported mediatype and a brief description of each resource of the CoAP server. The CoAPGateway uses such attributes to create its resources and assign them the given attributes.

This process is necessary to allow listeners to know what are the resources each node can serve. It is called every time a new node joins the network and periodically to control if the motes offer new resources or delete old ones.

The discover method of a CoAP client is a blocking operation as it is a CON messages that must wait for acknowledgment. Multiple variables in the medium could cause packet loss. CoAP has a mechanism of retransmissions using a default timeout and exponential back-off between each retransmission. Considering default transmission parameters, the maximum transmission wait arrives to 93 seconds. This time has been reduced for the discover method in order to allow other resources to be discovered. If the timeout runs out and the discover has not been completed, the same resource is enqueued with a lower priority in a sort of back-off fashion.

The WebLinks generated by the discover method are transmitted to the GatewayPublisher to allow MQTT clients to know the structure of each resource. The following snippet shows an example of the Erbium CoAP server to a discover request:

```
</.well-known/core>;
      ct = 40,
</sensors/light>;
      title = "Photosynthetic and solar light (supports JSON)";
      rt     =   "LightSensor",
</sensors/battery>;
      title = "Battery status";
      rt     =   "Battery",
</sensors/radio>;
      title = "RADIO: ?p = lqi | rssi";
      rt     =   "RadioSensor"
```

### CoAP server

Once all the available resources and its attributes have been loaded to *neighborsResource*, *coapServer* is instantiated, the resources are added and it starts on the only Endpoint available, 0.0.0.0/0. The port 5683, default for CoAP communications, must me opened in Kura Firewall in order to make the coapServer visible to all the CoAP clients in the local network.

*CoAPExchange* requests are handled directly by the Actuators or the Sensors Resources which forward the request to the proper mote. The response is delivered, as it is, from CoAP server to the clients.

### Resource service

CoAPGateway implements the SensorService interface of *GatewaySensors*. Therefore, it has to implement all the methods which allow the GatewayPublisher to access to sensor values and set actuators values. In addition, it must enable the publisher to access the other functionalities in order to retrieve information about the 6LoWPAN network and the motes resources. Another functionality offered to the publisher, typical of a publish/subscribe architecture, is the *observeResources* by which the listener can request a periodical update of a set of resources, indicating the root of interest. Accordingly, listeners can ask to stop such notifications.

The *ThreadRequestHandler* is the Runnable implementation that handles ResourceRequests, both periodic and non periodic. ThreadRequestHandlers are assigned to each mote. When the mote receives an observe call for the first time, it checks if its thread has been instantiated. If that is not the case, the assigned thread handler is scheduled at fixed rate in the thread loop.

Every time the ThreadRequestHandler gains access over the medium it executes a loop in which tries to manage all the requests to the motes contained in its priority queue. Once it has send it all or after a timeout, equals to two times the `COAP_MAX_TRANSMIT_WAIT` (186 seconds), it releases the medium and it updates its queue. Each periodic request is enqueued again at the end of the process, while normal requests are just discarded once they finish.

As previously mentioned in subsection 6.2.2, MQTT messages payload is represented in KuraPayload structures. Every time the GatewayPublisher makes a request over a sensor or actuator, it must specify the resource path, a KuraPayload and a pointer to itself in order to get the notifications back. The path is split to search for the resource along the tree. If any sub section of the path results not valid a BadRequest is raised and forwarded to the listener. The payload varies according to the type of resource.

### Data retrieval: getSensorValue() function

In the case of the *getSensorValue* call, it is possible to observe SensorResources or perform a normal GET request. This function has to parse the payload in order to understand what kind of communication is required. It supports both synchronous and asynchronous requests to the motes. Once it individuates the different possible variables in the payload it generates a ResourceRequest which is enqueued in one of the working threads PriorityBlockingQueue. A ResourceRequest object contains all the significative attributes of the publisher request, the interested CoAPResource and a priority level. GET requests have higher priority than OBSERVE requests. They are single specific resource request that have to be delivered as soon as possible whether observed resources will provide more values with a given period.

In order to observe a resource, a metric called "observe" must be added. Following the description of [12], the value of the observe attribute can be:

- **0 -** indicates that the listener wants to subscribe to that resource
- **1 -** indicates that the listener wants to unsubscribe from the resource

The update of the subscribed resources is executed periodically and the period between one reading and the other can be changed from the configurable interface of the CoAPGateway in the Kura web console. However, if a listener desires to receive the updates with a different frequency it can specify the attribute "publishPeriod" and the period length in milliseconds. Each time an observe get is called, the observe counter of the interested resource is incremented in a proportional way to the period duration in seconds. The counter is used by the listener to understand the order and the number of the readings.

As previously mentioned, observe calls are executed by the thread assigned to each MoteNode in the thread pool. When the first observe request arrives to the resource, a registration to the thread priority queue is made. Every time that another listener wants to observe the same resource it will be added to the resource list of listeners and automatically included in the observe process running within the dedicated thread. The resource will be observed until each listener unsubscribe from it.

When a normal GET request arrives, the service control if the value of the resource is still valid according to the publisher validityTime. In such case, this value is returned avoiding useless requests on the medium for data that is not time sensitive. If the lastUpdate of the resource is too old for that listener, a new request is necessary.

There are two cases for new GET requests. If the resource is being observed for other listeners, a ResourceRequest with the specific listener is created and enqueued within that observer for just one time. The ResourceRequest will be automatically

discarded from the queue as soon as the response is received. In the other hand, if there are no active observers for the MoteNode parent, a SimpleRequest is scheduled in the thread pool. SimpleRequest is another class implementing the Runnable interface that simply waits to get access to the medium and performs a single request. After which, the task is considered complete.

### Setting a mote resource: setActuatorValue() function

Its behaviour is similar to the getSensorValue() as it has to parse the KuraPayload to understand which kind of request it has to create. However, in this case, a payload with the desired value is needed. Once the required attributes of the KuraPayload have been validated, it proceeds to create a ResourceRequest with the proper method, PUT or POST. Such request is enqueued in the thread responsible of its Mote parent requests, if active. Otherwise, it creates a new SimpleRequest task in the thread pool.

### Neighborhood and neighbor information requests

As soon as the publisher is active, it calls the CoAPGateway to get information about the nodes in the 6LoWPAN network and their resources. The CoAPGateway answers with the information gathered by the RPL-DAG and Resource Discovery functions. Furthermore, it saves a reference to each publisher in order to send updated data every time it controls the sensor network.

### CoAPGateway configurable properties

Through Kura web UI is possible to modify a set of properties for the Gateway:

- *coap.maxTransmitWait*: Default timeout for responses in CON calls [ms]. CoAP default value is 93000.
- *coap.poolSize*: Default size of the thread pool.
- *coap.semaphore*: Default number of threads that can send simultaneous requests to the border router.
- *observe.sync*: Default type of connection for CoAP observe requests.
- *observe.rate*: Default rate of CoAP observe requests [ms].

## 6.3 AWS IoT

Among the different Cloud platforms that can be selected, AWS offers a wide variety of choices and possibilities and counts with a dedicated section for IoT devices. AWS IoT enables Internet-connected devices such as the Kura gateway to connect to the AWS Cloud and lets applications in the cloud interact with those devices. IoT applications can collect and process telemetry from the sensors and manipulate the actuators connected to the gateway.

Communication between a device and AWS IoT is secured by X.509 certificates that must be registered and activated with AWS IoT and copied into the device acting as a credential. Appendix section A.5 contains information regarding how to connect Kura with AWS cloud service. Once it is connected, the GatewayPublisher can publish and subscribe to topics in the cloud. AWS IoT counts with a MQTT client for testing purposes. Through this service it is possible to subscribe and publish on topics of interest and watch the resulting communication with the gateway.

AWS IoT devices use MQTT as communication protocol to the cloud. They publish their state in JSON messages to MQTT hierarchical topics names in order to associate the state to the device that is publishing it. Messages published on an MQTT topic are sent to the AWS IoT MQTT message broker, which is in charge of forwarding them to the topic subscribers.

There is a shadow thing for each connected device. This "shadow" stores state information which is later exposed in JSON documents. Items in the state information contains two entries: the last state *reported* by the device and the state *desired* by an application. It is particularly useful when dealing with intermittent connections. Webservers and application will always count with the last state and data of the device. On the other hand, an application may request a state change which generates an update in the desired field of the shadow and a message to the device. The device can synchronize with the shadow when it connects and report its new state.

The power of AWS IoT relies on the possibility of interconnecting the devices to all AWS services. This is possible through the definition of rules that filter messages and define a set of actions enabled when the rule matches. All the messages sent to the cloud and to other services are protected, authenticated and authorized by a configurable set of policies and security features. This versatility makes AWS a powerful choice to show IoT potential on data collection, analysis and consumption.

# Chapter 7

# Simulation and deployment results

This chapter describes all the tests that were conducted to evaluate the Gateway performance. The environment was first simulated in Cooja and then deployed on the Telosb motes. Different situations were considered, varying the distance between the motes, their quantity and the number of threads used in the gateway. The tests were conducted on the worst case scenario in which all the sensors connected in the WSN were being observed at the same time, at high rate.

Multiple factors were tested in order to understand which were the limits of the platform. The Gateway bundle was developed to allow full customization of factors as the maximum wait time for CoAP communications, the number of threads dedicated to the motes, as default is one for each mote. It is possible to define the observe rate for each thread and the number of threads that can send requests to the border router contemporaneously. The number of motes was gradually increased, starting from 1 CoAP server to a total of 9, equal to the number of available TelosB sensors.

### ObserveAll function

A special function was developed to test the worst case scenario. Through the *observeAll* method, a client in the cloud can request to start the test in which all resources will be added to their respective priority queue and all threads will start almost at the same time. A hundred messages will be sent to each mote distributed among its resources to test the responsiveness of the system. Each sensor resource will send a request until its node parent, the sensors node present in each MoteNode, has finished its tasks. After which, all successive requests for that mote will be omitted and discarded from the thread queue. All the response values are sent to the cloud in order to demonstrate the work of the service under pressure and to allow data analysis.

At the end of the observeAll function, a final message is sent, for each resource,

containing data about the performance. Number of sent packets, dropped packets, the time length of the CoAP request response and the total duration of the test are some of the elements present in the message.

#### AWS IoT MQTT client

After the device registration, a MQTT client is available for test purposes in AWS IoT console. The client allows to publish and subscribe to the interested topics (section 6.2.2). The gateway will send each requested CoAP message to the cloud and an update about the neighborhood and the WebLinks of each neighbor every 60 seconds. The GatewayPublisher subscribes to the /inbound topic in order to listen to cloud requests and the request of the AWS IoT MQTT client to start the test by giving the command action "OBSERVE$_A LL$".

## 7.1 Simulation

Cooja helps to better understand the performance of the Gateway. It allows to start the simulation in a noise free environment and to save time in mote programs installation and deployment. It also helps to simulate situations of multiple hops, since physical deployment becomes harder considering the motes could theoretically reach a distance of 200m with line of sight.

The CoAP servers used in the simulation contained three web resources: /light, /radio and /battery. All the motes were configured with CSMA/CA as MAC layer and nullRDC as radio duty cycle.

The length of each CoAP message was observed through Wireshark, a widely used protocol analyzer, in order to calculate the throughput for each resource. The length of the GET request packets from the CoAP clients varies according to the length of the resource name as seen in Fig 7.1, being the GET request for the /light resource the smallest with 84 bytes and the request for /radio?p=rssi the longest with 91 bytes.

Table 7.1: CoAPGateway configuration parameters and values

| Parameter | Value |
|---|---|
| **Observation rate** | 100 ms |
| **maxTransmitWait** | 93000 ms |
| **Semaphore access to the router** | 3 |
| **Pool size** | 1 thred for each mote |
| **Type of request** | Synchronous |

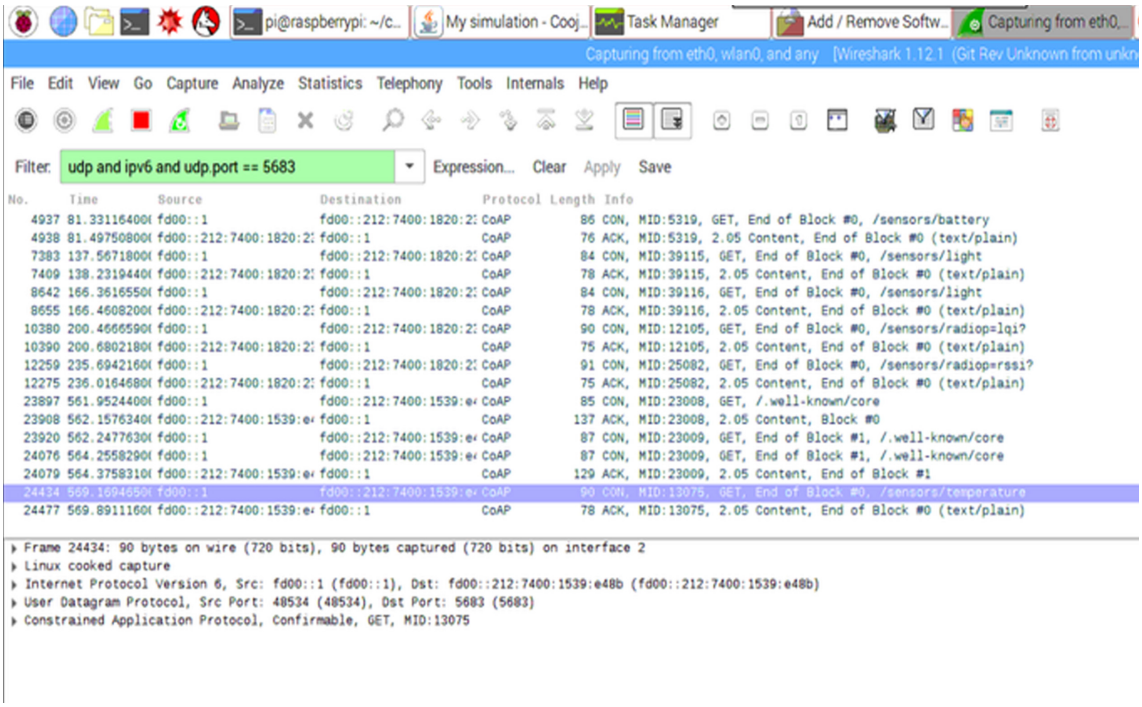



Figure 7.1: CoAP packets length

As previously mentioned, the CoAPGateway bundle can be configured to better adapt to different situations (Table 7.1). The timeout for the client requests was set, according to CoAP draft recommendations, to 93000 ms. In other words, the client will re transmit the message up to four times with exponential back off before considering the transmission lost. The scenario will test the gateway reliability and this setting enables CoAP best performance.

Synchronous calls provide better performance regarding the time of execution and enable a better understanding of what is the behaviour of the exchange. At the same time, it means the thread will remain blocked until it receives an acknowledge or the timeout runs out. For this reason, it becomes necessary to allow more threads to send requests to the border router. This factor can be tuned and, through experimentation, it was found that having a semaphore size equals to three, for more than 4 motes, provide a good trade off between blocking times and border router workload.

The requests were sent every 100 ms. Normally, data such as temperature, luminosity or battery levels would not be requested so often, a message every a few seconds or even minutes would be enough. But the purpose of this case scenario is to test the gateway under a big number of simultaneous requests.

A hundred requests will be generated for each mote, distributed among their web

resources. The packet transmission and reception rate will be configured at 100% to test the behaviour of the gateway under ideal conditions of noise and interference.

### 7.1.1 Simulation results

The simulation was focused on testing the reliability, the latency between CoAP client calls and the erbium CoAP servers response and the scalability of the gateway, both in number of motes and the distance between them.

The simulation runs on the Raspberry Pi, the border router is started through the tunslip6 tool and Cooja is connected to it through a serial socket server, as explained in the previous chapter.

Nodes were inserted into the simulation a few at a time, increasing the distance between them and the border router. In the final simulation, with 9 nodes, seen in Fig 7.2, the three is composed by 6 nodes in a distance of 1 hop and 3 in two hops. The simulations were conducted without altering the Tx/Rx ratio, in a space free of interference and noise. The results are shown in Table 7.2.

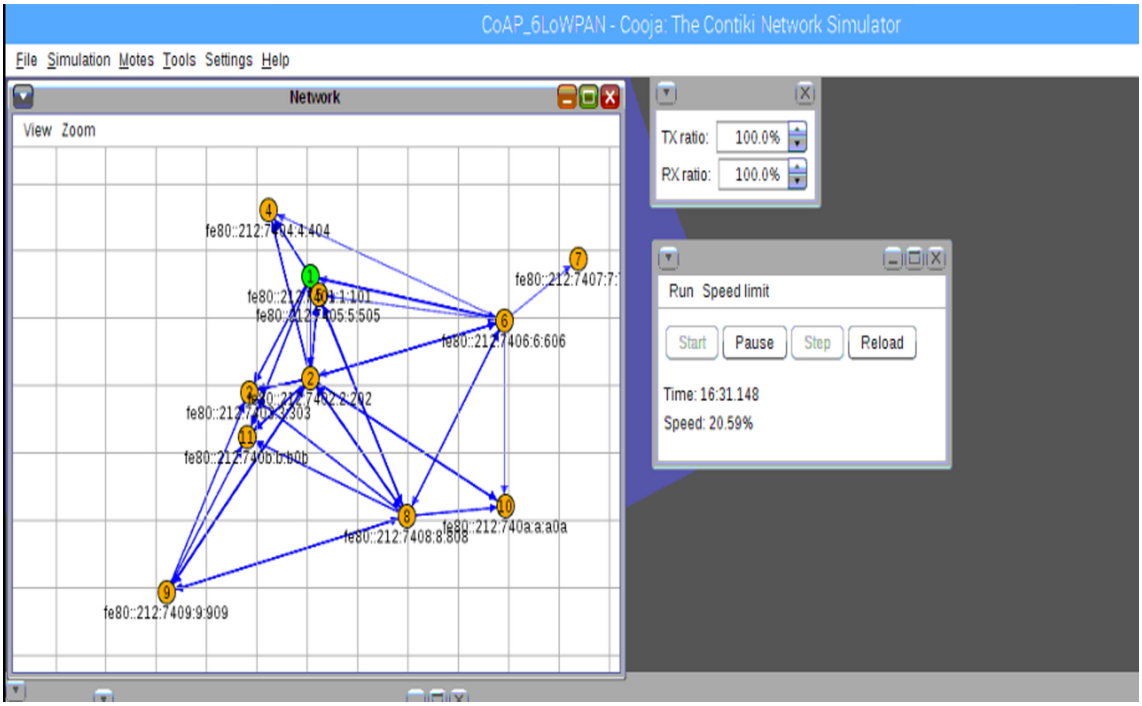



Figure 7.2: Final simulation structure

Being a noise free space, there is no packet loss. The attributes of interest are the average delay of communications between motes and the gateway, as well as the throughput, measured in Kbps. The throughput is calculated as:

$$\frac{100 * nMotes * 8(erResCoAP + gwResCoAP)}{\sum_1^{nMotes} \sum_1^{nRes} responsetime} \tag{7.1}$$

Where erResCoAP and gwResCoap are the average of the above mentioned values for the packet lengths sent by the gateway and the CoAP servers.

Table 7.2: Average throughput and average delay for the simulations

| **Number of motes** | **Average Throughput [Kbps]** | **Average Delay [ms]** |
|---|---|---|
| 1 | 4.78 | 258 |
| 2 | 4.28 | 290 |
| 3 | 4.25 | 296 |
| 5 | 2.87 | 436 |
| 8 | 1.88 | 677 |
| 9 | 1.83 | 688 |

It is possible to see that the throughput diminishes with every added mote but the result is still more than acceptable. It was observed that with a few motes, a better performance was obtained by reducing the number of working threads. However, it is not a practical solution in a real implementation due to the timeout, not for synchronous connections at least.

As expected, the nodes 2 hops away have a lower throughput and the delay time is almost twice respect to the rest of the less distant nodes. However, the throughput remains over 1 Kbps with a response slightly over a second for motes that are more than 50 meters away.

Table 7.3: My caption

| **Node** | **Average Delay [ms]** | **Average Throughput [Kbps]** |
|---|---|---|
| aaaa::212:7409:9:909/ | 722 | 1.74 |
| aaaa::212:7407:7:707/ | 827 | 1.52 |
| aaaa::212:7402:2:202/ | 604 | 2.09 |
| aaaa::212:740a:a:a0a/ | 1176 | 1.08 |
| aaaa::212:7403:3:303/ | 578 | 2.17 |
| aaaa::212:740b:b:b0b/ | 575 | 2.18 |
| aaaa::212:7404:4:404/ | 557 | 2.26 |
| aaaa::212:7406:6:606/ | 542 | 2.32 |
| aaaa::212:7408:8:808/ | 607 | 2.07 |

## 7.2 Physical deployment

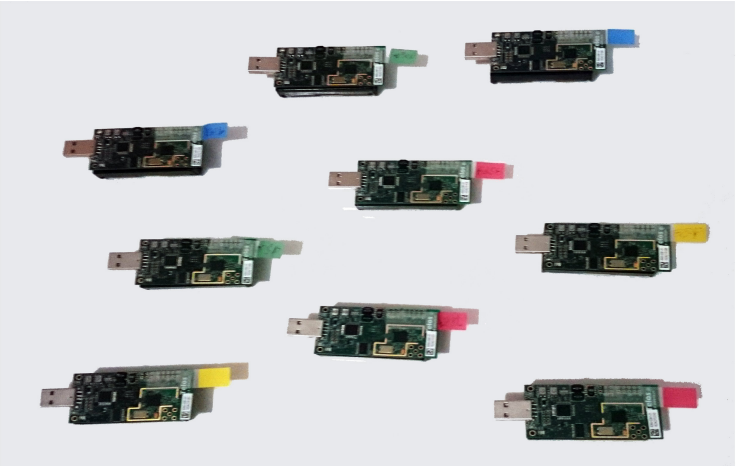


Figure 7.3: Deployed TelosB sensors

9 TelosB sensors (Fig 7.3) were loaded with different versions of the Erbium CoAP server in order to simulate a reality with multiple resources. The motes were distributed in the same floor, in different rooms, in a range of 20m around the border router connected to the Raspberry Pi. According to TelosB specifications, the devices should be able to communicate within a range of 40-50m indoor. The configuration of the motes is the same as in the simulations, CSMA/CA as MAC layer, nullRDC for the RDC layer, a thread for each mote and the semaphore size set to 3. The packets were generated, once again, every 100 ms for each thread.

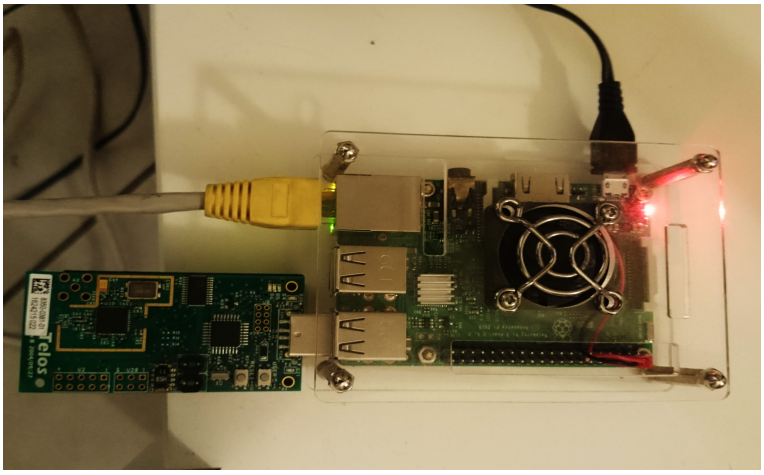


Figure 7.4: Gateway and RPL border router

The final set of web resources exposed by the servers is:

Table 7.4: My caption

| Number of motes | Temperature | Light | Radio | Leds | Battery |
|---|---|---|---|---|---|
| 3 | **X** | | | | |
| 2 | | **X** | | **X** | **X** |
| 2 | | | **X** | **X** | **X** |
| 2 | | **X** | **X** | | **X** |

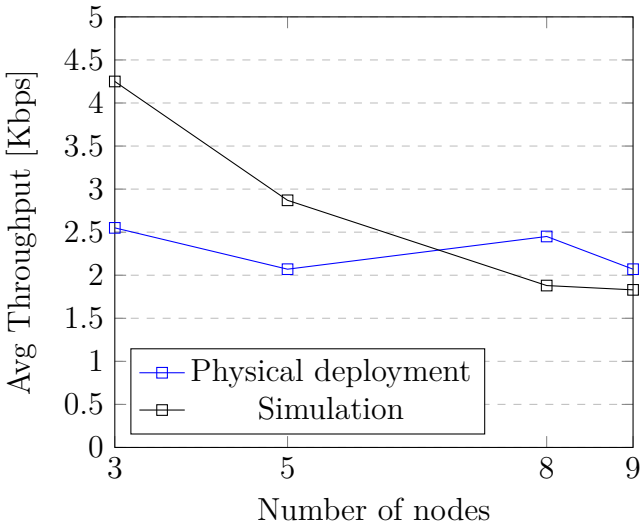
The temperature resource occupies a great portion of the mote memory, as it uses more libraries for the SHT11 sensor, and it has to loaded alone.

Table 7.5: Real deployment results

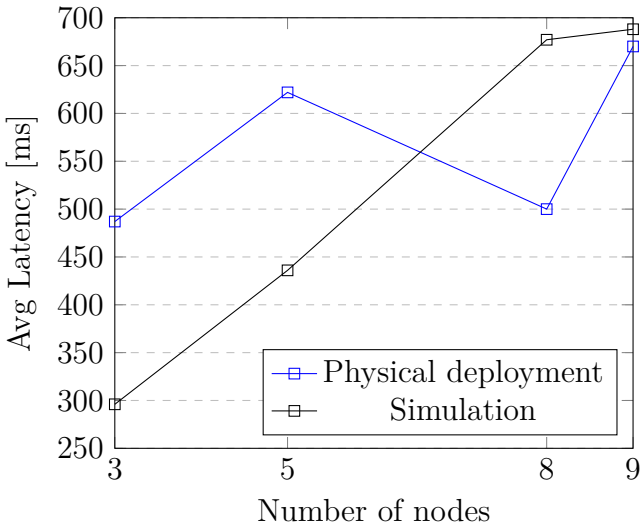
| Number of nodes | Avg Latency | Avg Throughput[Kbps] |
|---|---|---|
| 3 | 2.55 | 487 |
| 5 | 2.065 | 622 |
| 8 | 2.45 | 500 |
| 9 | 2.07 | 670 |

The results (Table 7.5) show that the gateway is reliable and performant even for 9 sensors. Comparative results with the simulations show better performances in the real deployment but mainly for the shorter distance between the nodes and the border router. In addition, the result of interference and noise is seen constantly, making possible that tests with 5 nodes perform worst than test with more motes. However the throughput remains always higher than 2 Kbps and the average latency lower than 700ms.

Average Throughput for simulations and physical deployment



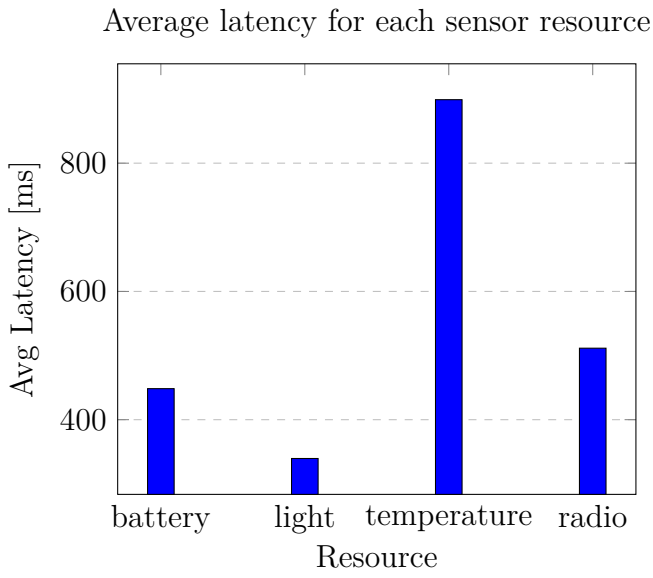

Average Latency for simulations and physical deployment




It was observed that the packets to the Cloud were sent with a considerable delay of almost 20s with a high rate of generated CoAP responses. The CloudService copies all the packets in a local database before the DataService publishes them. If a future application requires faster transmissions, the DataService should be implemented instead of CloudService. For continuous updates with fewer nodes, or with a frequency equals or higher than a second or more, message are received normally.

In the worst case with 9 motes, the average latency for sensor resource remains similar, around 500ms. An exception is the temperature resource that can last until a second with an average of 900ms. This behaviour can be seen in multiple tests and the main reason resides in the fact that TelosB takes more time to read the SHT11

sensor value. The average latency value for each resource is shown in the following graphic.




## 7.3 Integration with AWS

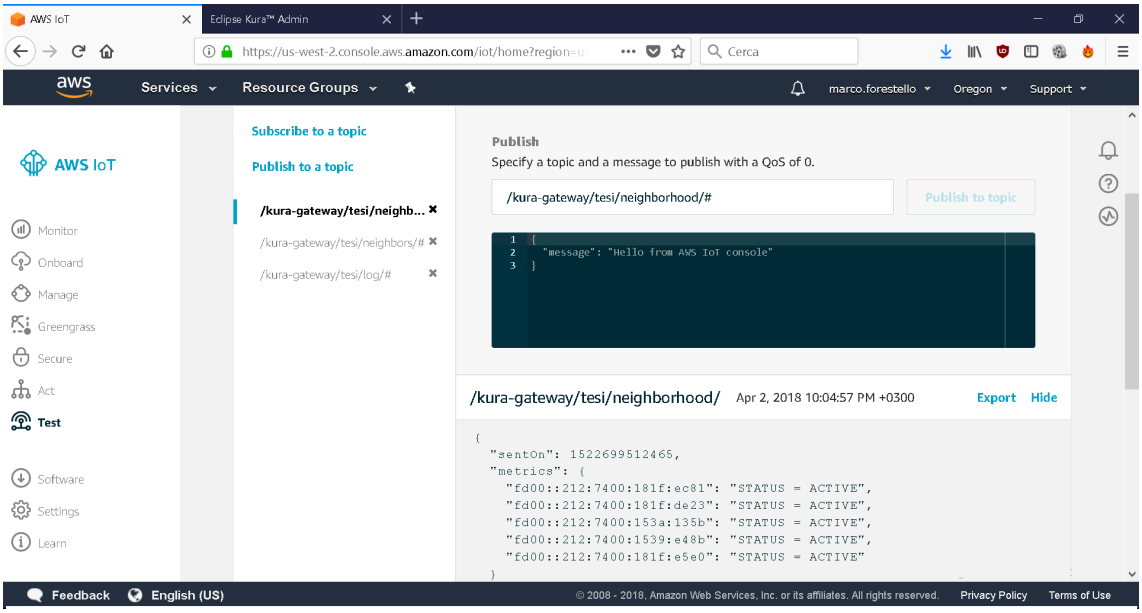



Figure 7.5: MQTTClient interface showing the topic /neighborhood

As previously mentioned, AWS IoT offers a MQTT client that can be used to test the communication with the gateway. It is possible to publish pseudo CoAP requests on the topic /inbound and to retrieve periodic information about the neighborhood subscribing to/neighborhoods topic. Mote WebLinks and pseudo CoAP responses can be found subscribing to topic neighbors/ and a more specific filtering can be made subscribing to a particular mote through neighbors/<neighborId>/ for the WebLinks and neighbors/<neighborId>/sensors/<sensorId> for the responses. An example of a normal GET request requested from the cloud is shown in Fig 7.6.

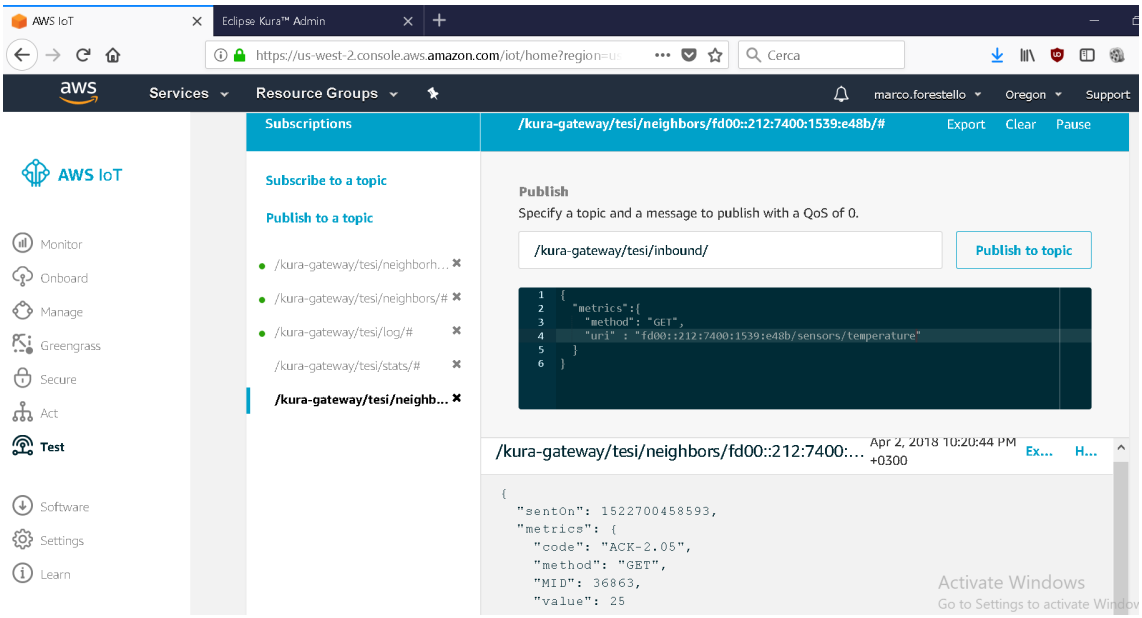



Figure 7.6: MQTT Client GET example

As soon as the WSN data reaches the cloud, it is possible to perform multiple operations to transform it, consume it or analyse it. AWS IoT offers the possibility of creating rules to filter the data and actions to call other AWS services. It is possible to create triggers for the data, such as an alarm call whenever a certain field is higher o lower than a threshold. The data can be sent through SMS, email or an application.

Data can be saved in DynamoDB tables to be later retrieved by Lambda functions. This is the principle of AWS serverless applications.

### 7.3.1 Data analysis on AWS

Sensors data is not particularly useful if not properly analysed. AWS allows to handle the constant stream of data coming from multiple devices. This data can be processed, analysed, and visualized in a scalable way that allow people to monitor

the performance, look for problems in the network and extract valuable information. Amazon Kinesis offers a practical an ease to use service for data analysis, a possible implementation is illustrated in Fig 7.7, extracted from AWS example [**?**].

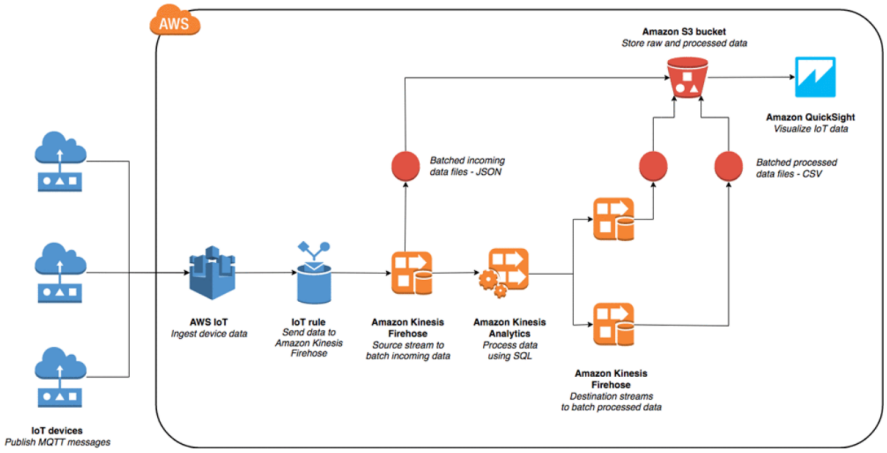



Figure 7.7: AWS IoT, Kinesis data analysis

Amazon Kinesis Firehose allows to capture, transform, and load streaming data from AWS IoT. Once the input is set through AWS IoT rules, it does not require further administration and it automatically scales to match the throughput of the stream data.

In a second step, it is possible to select data from the payloads and topics of the stream and process it with standard SQL through Amazon Kinesis Analytics. The processed data is forwarded to a Firehose delivery stream to consolidate the data stream in csv files allocated in S3 buckets.

The processed data is given to Amazon QuickSight, which is a fast, cloud-powered business analytics service that provides methods to easily create multiple data visualizations and perform data analysis.

The gateway was running for hours retrieving information of all sensor resources in the nodes every second. This data was fed into Amazon Kinesis Firehose dataT streams. A simple SQL query in an IoT rule is enough to select data of interest:

```
SELECT topic(5) as Node, topic(7) as Sensor, metrics.value as sensorData
FROM '/kura-gateway/tesi/neighbors/#'
```

The resulting stream can be processed in Kineses Analytics where another few lines of SQL code allow to create the rules for an output stream composed by all the sensors readings:

```
-- Create an output stream with four columns, which is used to send IoT
    data to the destination
CREATE OR REPLACE STREAM "DESTINATION_SQL_SENSOR_DATA_STREAM" (dateTime
    TIMESTAMP, moteAddress VARCHAR(25), sensorName VARCHAR(16),
    sensorValue SMALLINT);

-- Create a pump that continuously selects from the source stream and
    inserts it into the output data stream
CREATE OR REPLACE PUMP "STREAM_PUMP_1" AS INSERT INTO
    "DESTINATION_SQL_SENSOR_DATA_STREAM"

-- Filter specific columns from the source stream
SELECT STREAM "sentOn", "nodeAddress", "sensorName", "sensorValue" FROM
    "SOURCE_SQL_STREAM_001";
```

The resulting data tables are saved in multiple cvs files in folders of the S3 bucket separated by hour, day and month. Those tables can be added to QuickSight for different types of data analysis, some examples are shown below.

# Chapter 8

# Conclusions

The complexity in developing a gateway for the WSN and IoT resides in the wide variety of choices that can be taken at every step of the stack. Many decisions were made taking in consideration what could be the most appropriate development for a web resource using TelosB sensors as servers. The main objective of the thesis was accomplished. Through deep analysis about the existent technologies and protocols in IoT, it was developed a gateway solution with a specific stack allowing WSNs integration to the internet and to the cloud.

The gateway is able to communicate with any sensor acting as a CoAP server in an 6LoWPAN network having as a root the rpl border router, which is attached, in turn, to the Raspberry. CoAP provides an efficient, reliable and low powered RESTful architecture for constrained devices. The gateway gathers all the resources in the WSN. Such data can be discovered and requested both from CoAP and MQTT, two of the most used communication protocols for IoT.

The system allows multiple access to the data in a reliable, flexible and fully configurable way through the implementation of Kura platform and its web console. The gateway can handle at least 10 motes with a delay under a second at full capacity and CoAP retransmissions factors guaranteed zero dropped packets in all the tests, over an unreliable transport protocol as UDP.

Thanks to OSGi modularity, the created bundles can run along with other solutions. Furthermore, Kura was developed by Eurotech and the Eclipse IoT group, which guarantees a large number of companies and people that will help to further develop the platform. New solutions and new integration methods are being implemented as in the case of Apache Camel, an open source framework for message-oriented middleware with a rule-based routing that provides a Java implementation for the Enterprise Integration Patterns.

Amazon AWS was chosen as an example of the power of cloud platforms and to show how well they can connect to constrained devices such as the Raspberry Pi and platforms as Kura. The sensors on the motes were exposed by the gateway

and retrieved by the cloud as normal web resources that could be consumed in web applications or analysed on data streams. Kura can be connected to other cloud platforms and any MQTT broker implementing the MQTTDataTransport service.

### 8.0.1 Future work

Despite the good communicational features of the TelosB sensors, their main problem is the lack of memory. It is not feasible to use security features as DTLS over Contiki for this kind of mote. However, a 802.15.4 security layer, called LLSEC, can be implemented. It allows to encrypt and authenticate all the packets exchanged over the WSN medium using AES-32, 64 or 128. There are some limitations, mostly regarding anti-replay mechanisms but it remains a good solution for security within the WSN.

The tests were performed with RDC nullRDC. It means that every sensor was always awake waiting for radio communication. It is a good solution for testing and with motes powered by USB terminals but it is not feasible for long term reading with battery power. A duty cycle implementation is needed in order to set a sleep schedule for the motes. In addition, an interesting topic of research is CoCoA, a set of CoRE Congestion Control mechanisms. It is based on a retransmission timeout (RTO) algorithm that takes into account the round trip time estimates.

Regarding the gateway solution, a next step could be to establish a network of Raspberry Pis allowing multiple WSNs to communicate with each other through CoAP protocol and uploading readings results to a centralized database on the cloud.

Lastly, one of the reasons Contiki was chosen for is the possibility to deploy different modules on remote sensors. A future possible implementation could include this feature to allow to deploy different applications in the sensors from a remote location, whether in the Local network or over a Cloud application.

# Appendix A

# Appendix: SW codes and guides

## A.1 Contiki

### A.1.1 Contiki and Cooja installation

This guide contains the minimum configuration needed to implement Contiki for this project. [33] and [34] contains further information about the installation and development with Contiki.

Contiki must be installed in a Linux-based operative system. There are two ways to install Contiki:

- InstantContiki: it is a virtual machine prepared with all what it is needed to use Contiki.
- GitHub repository: Contiki can be downloaded from the repository. Additional components must be installed to properly run and compile all its features.

#### InstantContiki

InstantContiki is an entire Contiki development environment that runs on a Ubuntu virtual machine. It can be downloaded from Contiki's official web page `www.contiki-os.org/start` and installed in VMWare Player.

#### Github repository

In this case, a Linux-based OS is needed.

To install the toolchain and required dependencies (for TelosB), run in a terminal the following:

```
sudo apt-get install build-essential binutils-msp430 gcc-msp430
    msp430-libc msp430mcu mspdebug
sudo apt-get install openjdk-8-jdk opernjdk-8-jre ant libncurses5-dev
```

It is advisable to clone the latest Contiki git repository release (3.0 until now) to be updated with the last features and bug fixes. Contiki 3.0 is available at:

https://github.com/contiki-os/contiki/tree/release-3-0

In order to do so type:

```
sudo apt-get -y install git
git clone --recursive https://github.com/contiki-os/contiki.git
```

Alternatively, the repository can be downloaded and unzipped with [33]:

```
wget https://github.com/contiki-os/contiki.git
unzip 3.0.zip
mv contiki-3.0 contiki
```

### A.1.2 Mote programs installation

In order to compile and install these programs into a mote, connect the USB mote to the device having Contiki installed, give the permissions to write the mote:

```
sudo chmod 666 /dev/ttyUSB0
```

select, compile and upload the program choosing the mote type in TARGET and the usb port as in the following example for the border-router.c:

```
cd contiki/examples/ipv6/rpl-border-router
sudo make TARGET=sky border-router.upload
sudo make login (To view the mote output)
```

### A.1.3 Run Cooja

```
cd contiki/tools/cooja

ant run
```

## A.2 RPL border router

After having installed the border router (*contiki/examples/ipv6/rpl-border-router /border-router.c*) in a mote, the service can be started in two different ways:

- From the rpl-border-router folder in examples: once the physical mote is connected to the device, type:

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

sudo make connect-router
```

- From the tunslip folder: once the mote, whether physical or simulated, type:

```
cd contiki/tools/
make tunslip6
sudo ./tunslip6 -a 127.0.0.1 aaaa::1/64 (to make a connection
    between the RPL network and your local machine)
```

Once the Border router process starts it will show the IPv6 addresses of the border router from which it is possible to retrieve the addresses of all the motes in the network.

### A.2.1 Tunslip configuration for Cooja

In order to connect tunslip to the simulator it is necessary to add a rpl-border-router mote to the simulation. Then, select Tools -> Serial Socket(SERVER) and choose the border rooter among the simulation motes. The mote will start listening in port 60001 for client connections.

Such connection can be established as follows:

```
cd contiki/tools/

sudo ./tunslip6 -a 127.0.0.1 aaaa::1/64
```

## A.3 Raspberry Pi

### A.3.1 Raspbian installation

Raspberry Pi uses a dedicated linux-based operative system, Raspbian. Raspbian can be downloaded from the official site https://www.raspberrypi.org/

downloads/raspbian/ and loaded into a microSD card. In order to control the Pi, without all the peripherals, the ssh or the vnc server must be enabled. To enable SSH follow the instructions described in

https://www.raspberrypi.org/documentation/remote-access/ssh/

For the VNC server configuration follow these instructions:

https://www.realvnc.com/en/connect/docs/raspberry-pi.html.

Contiki must be installed, as explained in subsection A.1.1, in the Raspberry Pi in order to enable cooja and tunslip tools.

#### IPv6 addressing troubleshooting

**Setting raspbian in order to allow ipv6 addressing**

if **Error: SIOCSIFADDR: Permission denied** is seen after running tunslip6 it is necessary to delete the line:

```
net.ipv6.conf.all.disable_ipv6=1
```

from */etc/systemctl.conf* and to add the following lines:

```
net.ipv6.conf.all.autoconf=0
net.ipv6.conf.all.accept_ra=0
net.ipv6.conf.default.autoconf=0
net.ipv6.conf.default.accept_ra=0
```

## A.4 Kura

Kura offers a vast and clear documentation for installation, configuration and development [15].

### A.4.1 Installation

Kura was conceived having Raspberry Pi as one of its principal Hardware devices. The installation is well documented in the official kura documentation, under Raspberry Pi Quick Start, the link for the installation, the configuration and possible errors can be found in Appendix [ref appendix kura installation].

Once Kura is active and the Rasperry is wired both to Kura and to the LAN, it is possible to access Kura Web console [A.1] by inserting the Pi's address in the URL bar, the username and the password. This console exposes Kura's services and allows to change Network settings, Cloud settings and Firewall permissions, among others. The latter becomes important to open and forward different ports in order to allow a more pleasant control from a personal computer in the same LAN.

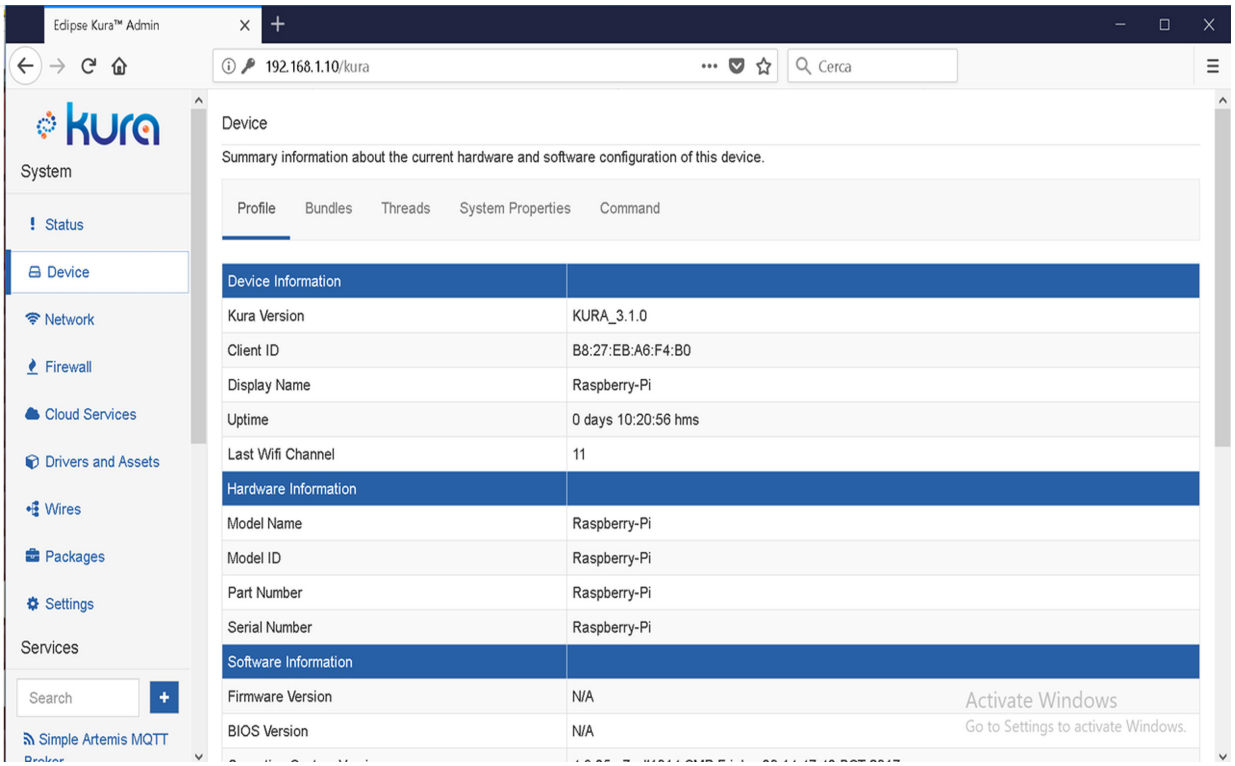



Figure A.1: Kura Web Console

### A.4.2 Development

In order to create Kura bundles it is necessary to install the JVM, EclipseIDE and download the Kura User Workspace archive https://www.eclipse.org/kura/downloads.php. A guide for doing so is available in [ref Kura documentation] under Development - getting started.

#### OSGi bundles

Bundles creation and deployment is described in [ref Kura documentation] under Development - Hello World Example and Deploying Bundles.

There are two ways to test the bundles:

- Local emulation mode (Linux/ OS X only)
- Remote Target Device: bundles can be deployed to the gateway as separated bundles or deployment packages. Deployment packages can be added from the Kura Console, under Packages → *Install/Upgrade*.

Plugins can also be uploaded to the remote device using the mToolkit plugin for

Eclipse, after having established a connection with the Raspberry Pi through its local address.

#### Kura remote debugging

It is possible to remote debug kura to better control and understand the environment situation. In order to do so, it is necessary to open the tcp port 8000 for localhost, to stop kura and to start the debugger by typing:

```
sudo systemctl stop kura

sudo /opt/eclipse/kura/bin/start_kura_debug.sh
```

Once it starts listening it is possible to test bundles by installing them through mToolKit, modifying the debug configuration to listen to the port 8000 and starting the debug. A more detailed guide can be found in [ref kura docs] under Development $\rightarrow Remotedebuggingontargetplatform$.

### A.4.3 Californium

Kura's main messaging protocol is MQTT but Californium can be imported as a jar into the project to enable CoAP communication. This project uses californium-osgi-1.0.0-SNAPSHOT.jar as the Californium library, it contains all the necessary libraries to run both the client and the server. The jar must be copied in the libs folder and added in the MANIFEST file under Runtime $\rightarrow Classpathtomakeitrun$.

A CoAP server will start by default at the 0.0.0.0/0 network endpoint. In order to allow other devices to access the server within the LAN it is necessary to open the port 5683 (default for CoAP communications) in Kura's Firewall for the address linked by Kura.

A brief introduction to CoAP programming with Californium can be found at https://docs.google.com/presentation/d/1dDZ7VTdjBZxnqcIt6qoX742d6dHbzap-D_H8Frf3LRE/edit#slide=id.p.

## A.5 AWS

In order to connect the Pi to AWS cloud follow the instructions of [15] under Cloud Platforms Connection -> Amazon AWS IoT platform. It describes how to register the Pi as an AWS thing and how to setup the security certificates and parameters to establish a secure connection. Once the device is connected to AWS it is possible to use AWS IoT console to manage it and to test MQTT messaging and rules.

# Bibliography


[1] K. Ashton. *That 'Internet of Things' Thing* in RFiD Journal, 2009.

[2] Cisco Systems, Inc. *The Internet of Things Reference Model* in World Forum, 2014.

[3] I. Lee & K. Lee. *The Internet of Things (IoT): Applications, investments, and challenges for enterprises* in Kelley School of Business, Indiana University, 2015.

[4] F. Bonomi, R. Milito, J. Zhu & S. Addepalli. *Fog Computing and Its Role in the Internet of Things* in San Jose, CA, USA, 2012.

[5] D. Gascón & A. Asín. *50 Sensor Applications for a Smarter World* http://www.libelium.com/resources/top_50_iot_sensor_applications_ranking/

[6] A.Dunkels. *FYI: Thingsquare mist* http://sourceforge.net/p/contiki/mailman/message/29876684/, 2012.

[7] B. Cody-Kenny, D. Guerin, D. Ennis, R. S. Carbajo, M. Huggard & C. Mc Goldrick. *Performance Evaluation of the 6LoWPAN protocol on MICAz and TelosB motes* in School of Computer Science and Statistics Trinity College Dublin, Dublin , Ireland, 2009.

[8] J. Polastre, R. Szewczyk & D. Culler. *Telos: Enabling Ultra-Low Power Wireless Research* in Computer Science Department University of California, Berkeley Berkeley, CA, 2005.

[9] E. Upton, R. Mullins, J. Lang & A. Mycroft. *RaspberryPi documentation* https://www.raspberrypi.org/documentation/

[10] J. W. Hui & D. E. Culler. *Extending IP to Low-Power, Wireless Personal Area Networks* in University of California, Berkeley, USA, 2008.

[11] IETF Z. Shelby, ARM, K. Hartke & C. Bormann. *The Constrained Application Protocol (CoAP)* in Universitaet Bremen TZI, 2014.

[12] CoRE Working group, K. Hartke. *Observing Resources in CoAP draft-ietf-core-observe-08* in Universitaet Bremen TZI, 2013.

[13] IBM Eurotech. *MQTT V3.1 Protocol Specification*

[14] D.J. Walker-Morgan. *Eclipse Kura-A Gateway for the Internet of Things* http://www.eclipse.org/community/eclipse_newsletter/2014/february/article3.php, 2014.

[15] Eurotech, D. Woodard.*Eclipse Kura: A gateway framework built for IoT* in

Virtual IoT Meetup, 2016.

[16] A. Bellin, P. Salandin & A. Rinaldo. *Simulation of dispersion in porous formations: Statistics, firstâ€order theories, convergence of computations* in Universitd di Trento, Trent, Italy, 1992.

[17] G. E. P Box & G. M. Jenkins. *Time Series Analysis: Forecasting and Control* in San Francisco, Holden-Day First edition, 1970.

[18] I. Butera & M. G. Tanda. *Acquiferi eterogenei soggetti a ricarica uniforme: analisi del campo di moto, dei processi di trasporto e condizionamento* in Tesi di dottorato, Politecnico di Torino, 1996.

[19] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill M. Welsh & E.Brewer. *TinyOS: An operating system for sensor networks* in Ambient intelligence, Springer, Berlin, 2005.

[20] A. Dunkels, B. Gronvall & T. Voigt. *Contiki: a Lightweight and Flexible Operating System for Tiny Networked Sensors* in Proceedings of the First IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, USA, 2004.

[21] V. Brussel, Freemind, MTP, Edosoft & MAIS consortium. *Contiki and Tiny OS* in Interoperable Sensor Networks, ITEA, 2012.

[22] A. Dunkels, O. Schmidt & T. Voigt, M. Ali. *Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems* in Proceedings of the Fourth ACM Conference on Embedded Networked Sensory Systems, Boulder, CO, USA, 2006.

[23] T. Reusing. *Comparison of Operating Systems TinyOS and Contiki* in Seminar: Sensorknoten - Betrieb, Netze Anwendungen SS2012 Lehrstuhl Netzarchitekturen und Netzdienste, Lehrstuhl Betriebssysteme und Systemarchitekturen FakultÃ¤t fÃ¼r Informatik, Technische UniversitÃ¤t MÃ¼nchen,2012.

[24] M. Kovatsch & J. Vermilliard. *Hands on with CoAP* in eclipsecon, France, 2014.

[25] Eclipse *Eclipse Paho Java Client* in https://www.eclipse.org/paho/clients/java/.

[26] IETF ROLL group. *RPL : IPv6 Routing Protocol for Low-power and Lossy Networks* in RFC6550, 2012.

[27] IETF Z. Shelby.*Constrained RESTful Environments (CoRE) Link Format* in RFC6690, 2012.

[28] M. Kovatsch, S. Duquennoy & A. Dunkels. *A Low-Power CoAP for Contiki* in IEEE 8th International Conference on Mobile Ad-hoc and Sensor Systems, 2011.

[29] M. Kovatsch. *Copper (Cu)* https://addons.mozilla.org/it/firefox/addon/copper-270430/

[30] Eclipse. *Eclipse Kura API documentation* in http://ftp.linux.org.tr/eclipse/kura/docs/api/0.7.0/index.html?org/eclipse/kura/message/class-use/KuraPayload.html

[31] ORACLE. *Java 8 documentation* in https://docs.oracle.com/javase/8/docs/

[32] ORACLE. *Java concurrency documentation* in https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

[33] USC. *Contiki tutorials* in anrg.usc.edu/contiki/index.php/Contiki_tutorials

[34] A. LiÃ±án*Colina*, *A.Vives*, *M.Zennaro*, *A.Bagula&E.Pietrosemoli.Internet of Things in 5 da*