NTNU
Norwegian University of
Science and Technology

Handover between Wireless Networks for Medical Applications

Michele Agostini

Master of Telematics - Communication Networks and Networked Services

Submission date:	August 2016
Supervisor:	Frank Alexander Krämer, ITEM
Co-supervisor:	Luca Spalazzi, UNIVPM
	David Palma, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Name of student: Michele Agostini

This project focuses on the medical area and how on the Internet of Things can provide an improvement. This could be an important improvement in the everyday life of patients, nowadays secluded in their locations, and an interesting step forward in use the new technologies in our lives.

The monitored data can be very different one from another, and different constraints may exist. It will be necessary to consider where the analysis is done, choosing between the sensor, the border router, the cloud or a combination of the previous options. Sensors can lose connectivity, so its required to offer a way to buffer data or analyze data directly in the sensor.

The main goal is to manage the handover between two different gateways, how and when it should happen, because of the time that it needs to be done. The signal power, the batteries or the work balance are some options that will be considered in the definition of the algorithm.

We assume multiple WPANs and search for suitable ways to let connected sensors, i.e. patients, free to move between them. This means manage the handover from one network to another always considering the constraints that the types of data monitored need.

Assignment given: 31th March 2016

Supervisor: David Palma

Responsible Professor: Frank Alexander Kraemer

ABSTRACT

This project takes inspiration from the medical area and focuses on how the Internet of Things can provide improvements. The idea is to build a system that leaves the patients move freely in the hospital and in the meanwhile to be able to properly manage data collected by their wearable sensors.

Therefore the main goal of the work is to manage the handover between two different WPANs. This is a key activity in these kind of systems, because it takes time to be performed and it is resources demanding.

The handover operation will be fully analysed and evaluated in all its characteristics in order to find a suitable solution for the project purposes.

To my family.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

6LOWPAN IPv6 over Low power Wireless Personal Area Networks

BLE Bluetooth low energy

CSMA Carrier Sense Multiple Access

DAG Directed acyclic graph

DIO DODAG Information Object

DIS DODAG Information Solicitation

DODAG Destination Oriented DAG

FFD Full Function device

HTTP Hypertext Transfer Protocol

IEEE Institute of Electrical and Electronic Engineers

IETF Internet Engineering Task Force

IOT Internet of Things

IP Internet Protocol

IPV4 Internet Protocol version 4

IPV6 Internet Protocol version 6

ISM Industrial, Scientific and Medical

JSON JavaScript Object Notation

MAC Medium Access Control

MCU Mobile Control Unit

MQTT MQ Telemetry Transport

OF Objective Function

OS Operating system

OSI Open Systems Interconnection

QOS	Quality of Service
RAM	Random Access Memory
RDC	Radio Duty Cycle
RFD	Reduced Function Device
RPL	Routing Protocol for Low-Power and Lossy Networks
RSSI	Received signal strength indication
SLAAC	Stateless address autoconfiguration
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USB	Universal Serial Bus
WPAN	Wireless personal area network

1 INTRODUCTION

1.1 MOTIVATION

The increasing number of connected systems that surround us and the rising of new technologies such as the Internet of Things and the Cloud Computing are changing the World. It is now possible to think of new ways to reinvent every aspect of our lives and this project aims to contribute this change.

If we imagine a hospital where there are many patients, each one of them wears several sensors that measure various parameters. Nowadays they are wired at their locations, but why do not we imagine to free patients from wires, make them comfortable and, above all, allow them free to move within the hospital without bonds?

This means creating a communication infrastructure consisting of:

- patient sensors
- gateways
- routers
- cloud

Patients will be monitored by wearable sensors connected to gateways and routers located in the hospital area and a cloud service in which all the data can be collected and analysed.

The medical aspect is out of the scope, the parameters are not defined, but it is clear the fact that they will be of a different nature and therefore with different needs and management. The body temperature sensor will need a lower transmission rate than heartbeat sensor. The latter will need to be considered essential, you can not waste any of the available data.

This situation opens big issues and challenging ideas, there are a lot of options and questions to answer to: Which technologies? Which infrastructures? Which platforms?

What happens if a sensor loses communication signal to the gateway: will it have a buffer or will it be able to analyse data itself? How do we handle the handover between the various gateways? Depending on what do we choose?
The roaming of the sensors through the infrastructure will be a key aspect of all the project, it will be fully analysed in this work.

Another key aspect of the project is the resource constraints having sensors that will be powered by batteries. It is necessary to take into account the consumption of resources in every aspect: in the technology to be used, how to handle analysis, how to manage handovers, which micro controllers to choose.

1.2 SCOPE AND OBJECTIVES

1.2.1 *Scope*

This project is to focus on the Handover of a node between networks, since this is a key aspect of the Internet of Things technologies considering our scenario.

In order to properly manage the handover, we will need to:

- see how an handover works in the standard implementation
- understand which layers and variables are involved in this operation
- evaluate the performance
- define if improvements are needed for our own purpose

To discuss these points we will provide details about the implementation of the network and backgrounds on the involved technologies.

1.2.2 *Objectives*

O.1: Build a network

The first step of this project will be to implement a network for our test, without this step would be impossible to continue the work.

O.2: Understand the network

The handover is a sensitive operation that involves most of the mechanisms of the network, so we need a deep understanding of the system to fulfil our purposes.

O.3: Evaluate the default handover implementation

To understand if the default implementation of the handover is suitable for our purposes we need to investigate its behaviour and test it in order to have all the information about it.

O.4: Improve the handover

The last objective of this thesis is to improve the aspects that not satisfy of the handover mechanism so we will finally have an implemented network with a working handover system that will let future projects continue to work in this subject area.

1.2.2.1 *Research questions*

R1: Is IEEE 802.15.4 a good choice to support the structure we need?

We answer this question setting up a network and testing its performance with different upper layers options.

R2: Is MQTT suitable to be on the top layer of our network?

On the upper layers the options are multiple, but MQTT seems one of the most used options. We will test the MQTT behaviour during all the path of this project.

R3: How is the Handover handled in the default implementation?

First thing we need to do to achieve our goal is to study the default implementation of the handover, understand it evaluate it.

R:4 How is possible to improve the Handover considering our purposes?

The answer to this question is strongly dependent on R3, because only after the full comprehension of the standard implementation will we be able to know where and how we can do our improvements.

1.3 METHODOLOGY

The work we describe in this thesis can be methodologically divided in three main parts.

In the first one we describe how we implement a working network that uses MQTT and we provide the details necessary to face a connection issue shown by the system.

In the second part we start to investigate the handover mechanism, we understand which part of the system are involved and we find out some problems that make the standard handover not enough performing for our purposes.

In the third and last we show our proposals to improve the standard handover mechanism.

1.4 STRUCTURE

Chapter 2 talks about the background knowledge that is needed to go through this project, speaking about both hardware and software technologies.

Chapter 3 describes the implementation of the network that we will use for the following work. After the implementation we describe how the network works and how it is possible to solve a connection issue about the Radio Duty Cycle that made the connection phase too long. This chapter answers to objectives O.1 and O.2 and gives information about R.1 and R2.

Chapter 4 goes deep in the handover mechanism. It shows what an handover is and how it takes place in the standard implementation of the network: testing it, analysing it and identifying the most important issues to solve to make improvements. With these information we discuss O.3 and R.3.

Chapter 5 illustrates the proposals we make to improve the handover mechanism, the reasons why we think these different versions improve the system and the results of the test we did to show their performances. This chapter is about O.4 and R.4.

Chapter 6 discusses the future work and summarizes the work done with the conclusions. With this chapter we complete the answers to R.1 and R.2.

2 STATE OF THE ART

In this chapter we present the main technologies that will be analysed in the work. Further details will be presented during the work too, in order to make it easier to understand.

2.1 IEEE 802.15.4

IEEE 802.15.4 [1] is a standard defined in the 2003, it provides the physical layer and the media access control of the OSI model. Offering the lower network layers of the OSI model, it is meant to be an operational base for any system that focuses in low power and low cost characteristics, this means also low data rate and low range cover.

These characteristics make IEEE 802.15.4 slightly different from Wi-Fi [2] technology that is meant for large bandwidth and more power applications.

The physical layer (PHY) provides the actual transmission of the data and the medium access control (MAC) layer let the frames from the upper layers to go through the physical layer.

IEEE 802.15.4 supports two types of network topologies: star and peer-to-peer, and the nodes can be of two types too: Full Function device (FFD) and Reduced Function Device (RFD). The former, can do any operation, the latter are simpler and can only communicate with the FFD nodes.

The definition and the options for the upper layers is up to the developer, several projects are based on this technology due to the increasing interest in the IoT.

2.2 BLUETOOTH LOW ENERGY

BLE [3], acronym for Bluetooth low energy or Bluetooth LE, is a wireless technology standard developed by Bluetooth Special Interest Group. Even if it shares part of the name with his famous brother (classic) Bluetooth, it is quite different.

BLE strongly focuses on low power, low cost and low requirements systems, but it manages to achieve the same communication range. In fact the BLE has a data rate up to 1 Mbit/s, a power consumption up to 0.01-0.5 W and almost the same range of classic bluetooth about 100 m.

BLE and Classic Bluetooth share the same spectrum range too: 2.400 GHz-2.4835 GHz ISM band, but they have different channels.

Since the Bluetooth 4.0 specification is it possible for devices to provide both BLE and Classic Bluetooth implementation (it was not back compatible before), this has been a great step forward for the diffusion of this technology. Nowadays it is supported by a large number of hardware manufactures and Operating Systems.

2.3 IPV6

IPv6 (Internet Protocol version 6[4]) is the newest IP (Internet Protocol) developed by the Internet Engineering Task Force (IETF), it has become necessary to outdo the old IPv4 that was not ready to face the new technology challenges.

IPv6 was not designed to be compatible with IPv4, but a lot of systems of communication have been developed and it is now possible.

One of the differences between the two versions is the number of the addresses, IPv6 make possible to have an enormous amount of addresses thanks to the different number of bits used: 128 bit long addresses replace the 32 bit old ones. This means something about 2^{128} options or $3.4x10^{38}$ addresses.

The larger number of addresses is not the only difference between the two protocols, other improvements are carried out by the IPv6:

- hierarchical address allocation
- multicasting that enables the transmission of a packet through multiple destinations at a time
- Stateless address autoconfiguration (SLAAC) and Neighbor Discovery Protocol that permits to a node to auto configure itself in a network
- routers processing simplified

The structure of an IPv6 packet consists of two parts: a header and a payload, but optional headers can be added to enable special information. Nowadays the number of connected smart systems is increasing very fast and IPv6 is an essential step forward to permit the development of new technologies.

2.4 6LOWPAN

6LoWPAN is the acronym of IPv6 over Low power Wireless Personal Area Networks[6], it has been developed by the IETF to answer the new technological needs. The raising of the Internet of Things showed how important is to have a system to create networks between low power and low resources devices.

The main goal of 6LoWPAN is to be an adaptation layer between the wide spread IPv6 and the IEEE 802.15.4, which can be read as connect internet to the world of small resources systems.

There were some important issues to connect these worlds, one of the most important was the different size of the packets between the two layers. In order to solve this problem 6LoWPAN provides a encapsulation and header compression system that allows the two worlds to communicate.

Other important features of 6LoWPAN are the address autoconfiguration, that makes possible to create IPv6 stateless addresses, the mesh routing protocol, to use the multi-hop technology, and the use of the 15.4 frames in which the IPv6 packets are fragmented.

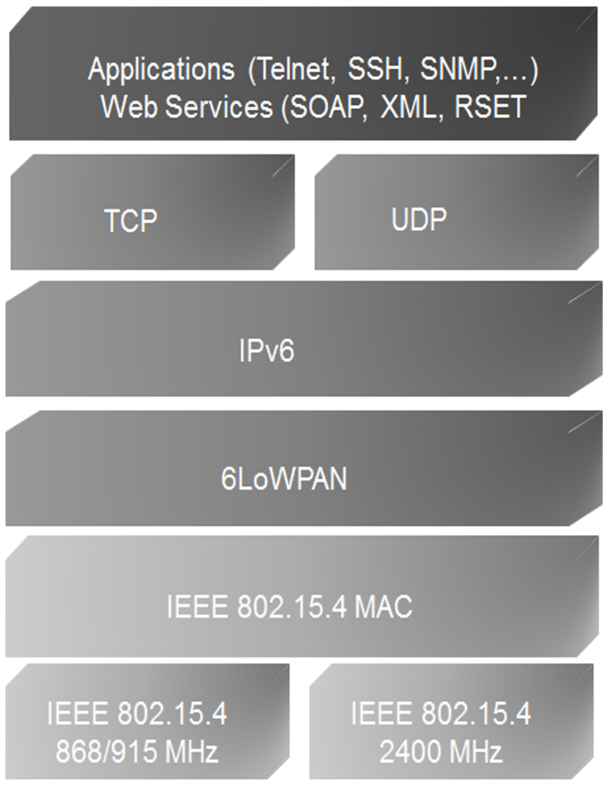


Figure 1: 6LoWPAN in the protocol stack (source: IoTin5days[5])

2.5 RPL

RPL means Routing Protocol for Low-Power and Lossy Networks [7] and it is based on a distance vector routing specifically designed for systems with resources constraints, developed by the IETF group.

The image below, taken from the IoTin5Days document, shows the role of RPL in the protocols stack:
The RPL network created is a Destination Oriented Directed Acyclic Graph (DODAG), it supports different traffic patterns and it is identified by an id, the DODAGID.

In a single network we can have different RPL instances, which can contain multiple dodags; a node can be part of multiple instances but it can be contained in only one DODAG for every instance.

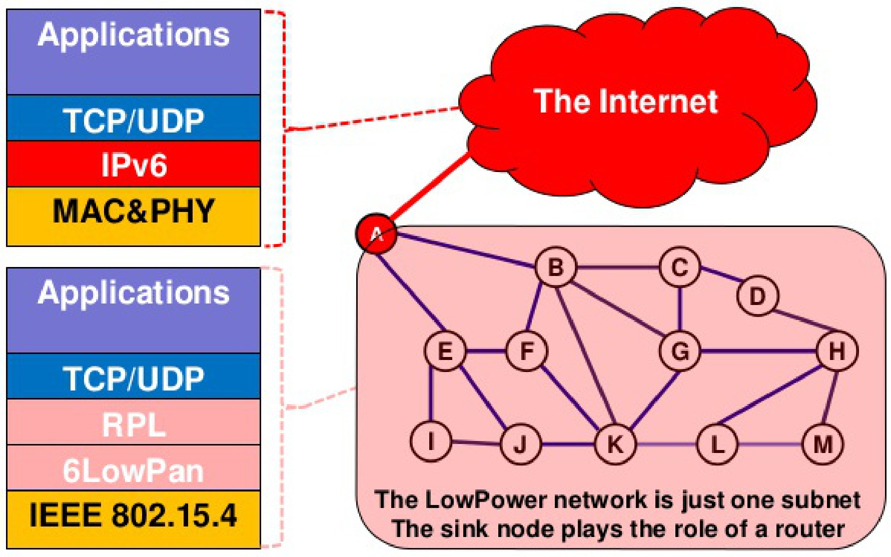


Figure 2: RPL in the protocol stack (source: IoTin5days[5])

Every DODAG has a DODAG root that acts like a sink, every path from the other nodes ends to it, because the graph is acyclic.

The connection phase
In the scenario of this project more than the routing protocol itself, that is optimized by the Objective Function (OF) and the Rank number, we are interested at the connection establishment that has a key role in the handover happening.

If we consider that we have a working DODAG, the connection phase of a new node starts after the bootstrap with the broadcasting of a DODAG Information Solicitation (DIS) message, it is used to request information about the available DODAG to other nodes (not necessarily the root).

A node that receives a DIS answers to the sender with a DODAG Information Object (DIO), a message that contains information about the DODAG.

Once the new node receives a valid DIO, it is processed and the node can join the network.

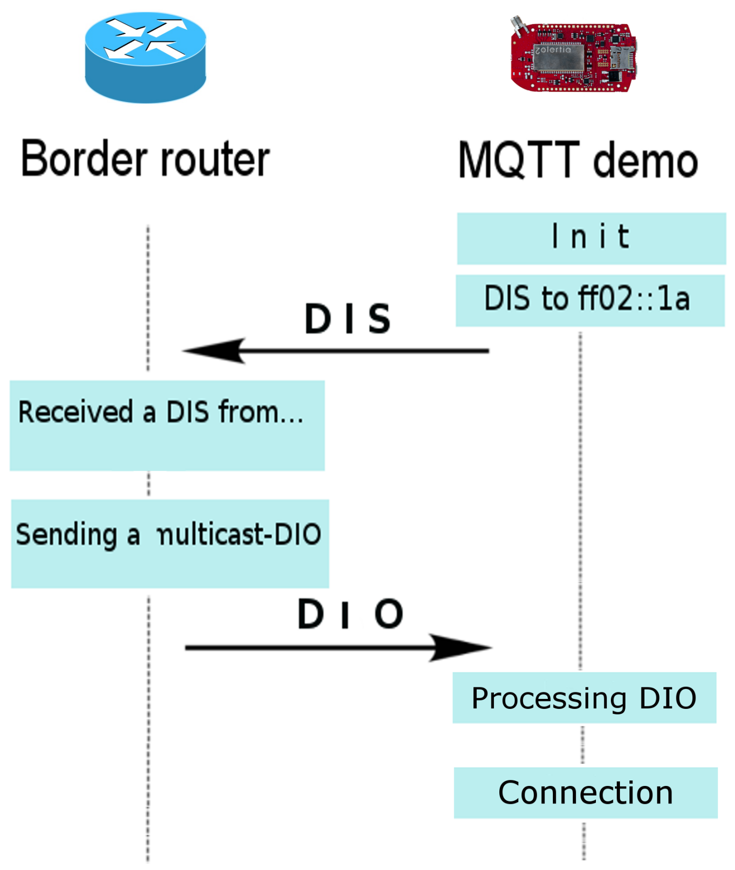


Figure 3: RPL connection scheme

2.6 CONTIKI

Contiki [8] is an open source operative system developed by a worldwide community. It is designed to be very light and low resources demanding so it can run over several low power systems. It is in particular good to be used in embedded semi-controller with few hardware resources.

It is based on the C language and processes are designed as protothreads, this lets the developers fast implement their projects using the wide known thread programming technique.

Despite its lightness, Contiki is full of useful features:

- it is a multitasking OS
- it is memory efficient

- it is power efficient
- it is it possible to load modules in run-time
- it has a network simulator called Cooja, that let the developer fast simulate large networks

Contiki has another main characteristic that makes it very interesting for the purpose of this work, it is designed specifically the Internet of Things. This means that, in addition at the essentials features seen above, the IP stack is fully implemented: IPv6, UDP, TCP, HTTP and 6LowPAN.

2.7 THREAD

Thread is a protocol for the Internet of Things developed by the Thread Group[9], in which it is possible to find an interesting number of famous companies such as: NXP, Samsung and ARM for example.

Thread is IPv6 based and this make it very compatible with other systems, the Thread stack provides the Transport and Network layers of the OSI model and it is designed using other standards as UDP (User Datagram Protocol) transport and 6LowPAN with IPv6 addressing. It is based in IEEE 802.15.4 for the lower layers, and it gives to the developer total freedom for the application layer.

As it is possible to see, Thread is not a whole new standard, but it has

- Simple commissioning to join or leave a network
- Power saving support for sleeping devices
- Full point to point network without single point of failure

The Thread network is based on 4 characters connected with thread links:

- Border Router: it connects the network to the cloud
- Leader: the coordinator of the network
- Thread Router: it is able to become leader if necessary
- End Device Router Eligible: it can be promoted to router if required by the leader to improve the network

2.8 MQTT

MQTT[10] (MQ Telemetry Transport) is a connectivity protocol that works over the TCP/IP and it has been developed to work with low resources consumption.

It is based on the Broker architecture, the messages are organized in topics with the publish/subscribe model. A node

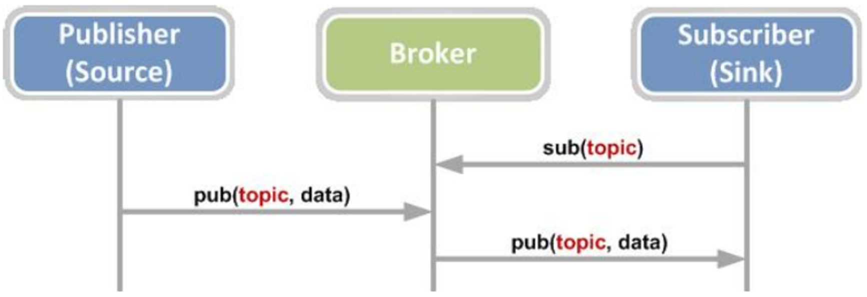


Figure 4: Broker architecture in MQTT (source: IoTin5days[5])

publishes its messages to its publish topic and will receive from the Broker any message published in its subscribed topics.

Different Quality of Service (QoS) levels are available that define how the messages are delivered to other nodes.

The default Json (JavaScript Object Notation) format for the messages makes simple the combination of this technology with other systems.

Mosquitto
Mosquitto[11] is a MQTT broker developed by the Eclipse IoT Working Group. It is OpenSource and it is specifically thought for the Internet of Things and this is why it is has been chosen for this project.

2.9 ZOLERTIA

Zolertia[13] is a company located in Barcelona that produces systems for the Internet of Things. Their projects claim to be appreciated both for enterprise solutions and the academic or developers worlds. Zolertia produces its own hardwares and firmwares or they design specific solutions for their client.

At the moment Zolertia produces three main platforms:

- **RE-mote**: it is the most complete development board produced by Zolertia. It carries two radios, battery charger, external storage, many interfaces and connectors
- **Firefly**: it is a small and simple card that carries only the essentials features, but this let it be used in many scenarios and it can be extended
- **Z1**: is a general purpose card with a low power MCU, a 2.4GHz Transceiver and sensors

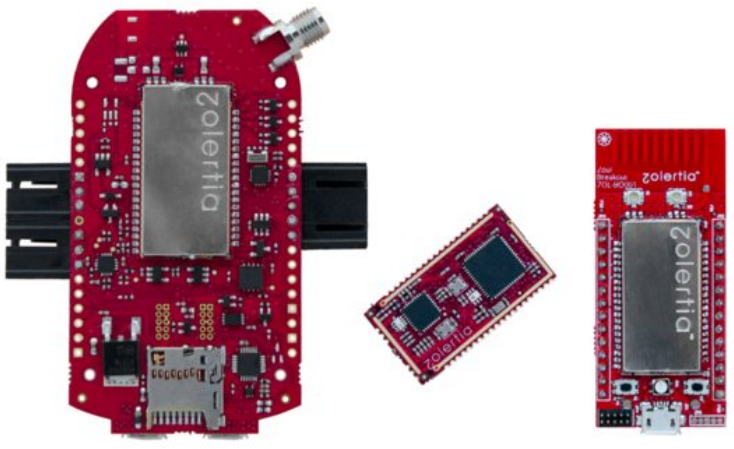


Figure 5: The Zolertia boards (source: Zolertia[12])

The first two platforms, RE-mote and Firefly, are based on the Zoul module. It carries a CC2538 core by Texas Instruments, an ARM Cortex, 32Kb RAM and double RF interface.

Zolertia platforms support different operating systems, among which Contiki OS and RIOT: an OS for low memory systems with real-time functions.

3

IMPLEMENTATION OF THE NETWORK

In this chapter we describe the first part of the practical work of this thesis that goes from the classic *Hello World* example to the first implementation of a working network.

The work described in this chapter used the document IoT in Five days [5] as useful way to go through these first steps. Before to start the work a whole reading and understanding of this guide has been necessary and it has been the base of this first stage of the work.

Despite the early stage of these steps and their easy-go first looking, they required time to be well implemented and understood.

3.1 HANDS ON

The work described in this section includes the first steps needed to familiarize with the Zolertia and Contiki platforms, from the installation until the first communication examples.

3.1.1 *Hello World*

As in every project the work starts with the installation of the platform Contiki in our host.

After the resolution of some missing dependencies and some needed update in the guide, we are ready for the classic *Hello World* example that establish the starting point of the work.

This first step let us understand how the communication between the mote, a Zolertia Re-mote, and our host works. The *Hello World* example simply write in the mote shell a corresponding message after the mote bootstrap phase.

The next step is to analyse a contiki process thread that has the following structure:

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

   printf("Hello, world\n");

   PROCESS_END();
 }
```

[14]

After this first example we moved forward exploring the Contiki OS and the Re-mote mote features. We learned to use Contiki events, as a buttons pressure, handle the board leds and reading values from the board multiple sensors.

With these first examples and with a deepening of the Contiki platform we are ready to enter the area of work we are interested in, the communication between motes.

3.1.2 *Wireless with Contiki*

This chapter marks the beginning of the work on the communication between motes and it is based on 3 different roles: a border router, a client and a sniffer to see and analyse the packets traffic.

To set up our first beginner network we used the follow motes organization:

- mote1: rpl-border-router example
- mote2: udp-client example
- mote3: sniffer example[15]

The **Border Router** is the conjunction between the external world and our 6LoWPAN network based on the IEEE 802.15.4 protocol. The packets are handled by the RPL and directed to

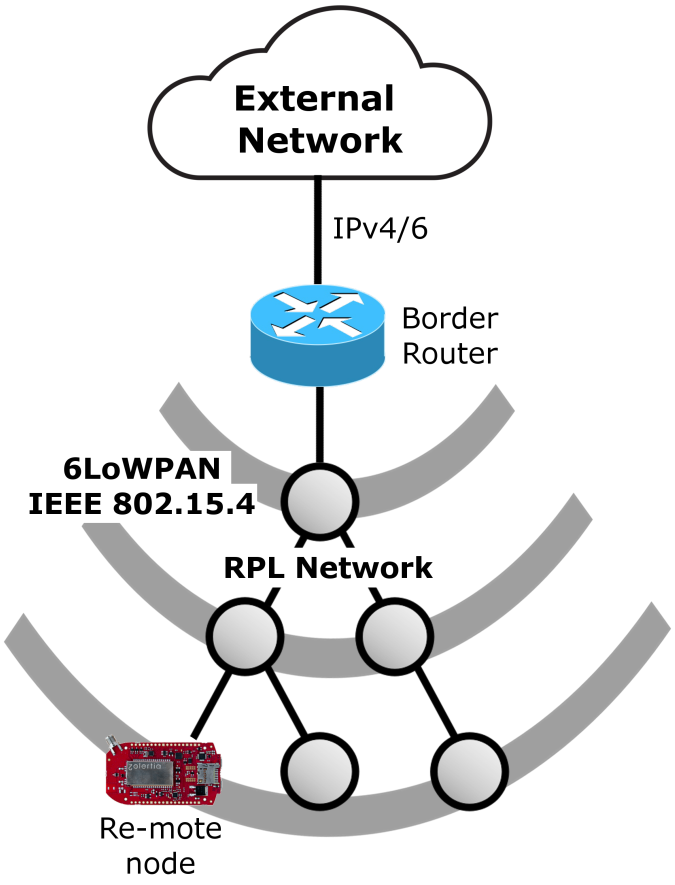


Figure 6: Network example

the DODAG root, the border router.

To connect the border router to the external world in this project we use the Tunslip utility provided in Contiki. **Tunslip** creates a bridge that connects the border router and the RPL network to the local machine.

Tunslip is activated in the border router *Makefile* when we use the command:

```
make connect-router
```

to start the border router.

On top of that stack many technologies can be used, both UDP as in this example and the TCP protocols are supported, like the messaging application MQTT.

In this first implementation the udp-client example simply broadcasts packets that are redirected to the DODAG root and then can be processed and in case sent to external applications.

In order to fully understand how the network works even other udp examples have been examined, the udp-client will be the one we will use during the rest of project as test for its easy way of working.

The sniffer is used to sniff packets as its name explains and it can be used combined with softwares like Wireshark[16] to analyse the packets traffic.

Meanwhile the implementation of the first two examples did not require many efforts, it can be useful describe the implementation of the sniffer. In fact we have to use another application, Sensniff, to connect the two extremes of the pipe: Wireshark and the sniffer.

Important to note is that at the moment this project is being done it has been discovered a bug that does not let use the sniffer example with the Re-mote board. This made necessary the use of a supplementary Z1 board.

After some settings optimization work in Wireshark, like the *"dissect only good FCS packets"* option, it is possible in the end to see the packets traffic and it is very useful to comprehend how the system works and to debug future problems.

3.2 SET UP OF A MQTT NETWORK

After these first steps useful to understand the basics of the platforms used in this project, in this chapter we introduce a new technology that is very important in the final implementation: MQTT [10].

In this chapter we illustrate all the characters participating in the system and we implement our first MQTT network.

3.2.1 *MQTT, Mosquitto and MQTT.fx*

As explained in the second chapter a MQTT network is composed by clients and a broker.

The MQTT clients are based on the *mqtt-demo* example available in the Contiki platform, instead as broker we chose to use the open-source Mosquitto broker[11] installed in our host.

This structure as to be paired of course with a border router in order to have the communication between the MQTT client and the broker through 15.4.

There is one thing that we need to pay attention to, the broker default listening port. This is because in the *mqtt-demo* example the default port is set to 1883, so we have to be sure that the MQTT client port and the broker port will be the same in order to have communication. To change the listening port of the broker we can use the *-p [port number]* option.

After this we witnessed the first connection, made clear by these broker messages:

```
New connection from aaaa::212:4b00:616:fc9 on port 1884.
New client connected from aaaa::212:4b00:616:fc9 as
    d:mqtt-demo:cc2538:00124b160fc9 (c1, k11520,
    u'use-token-auth').
```

It is possible to note as the MQTT client id is build up from the node address.

The connection goes through 3 phases made clear by the node led, as well explained in the example *Makefile*:

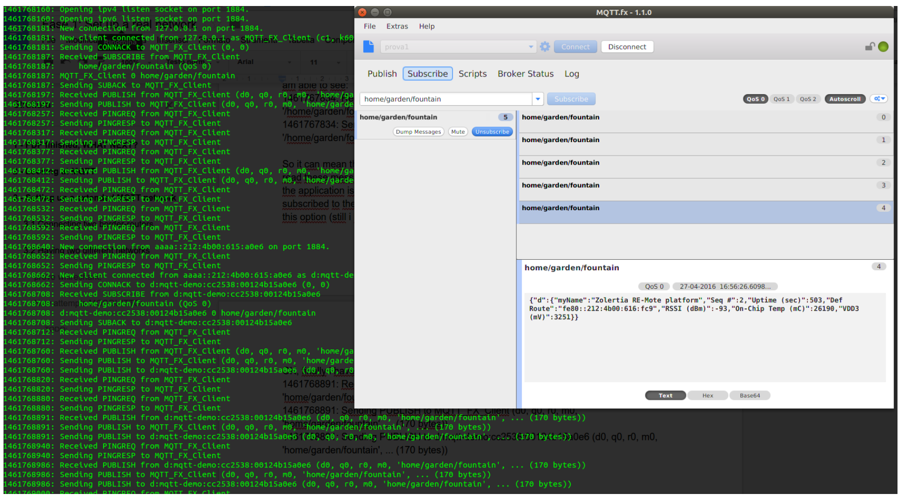
- fast blinking led: searching for a network
- medium blinking led: connecting to a network
- slow and long led: connected / publishing

3.2.1.1 *MQTT.fx*

MQTT.fx[17] is a desktop application that works like a MQTT client and it is very useful to connect to the MQTT broker and read/send messages.

In MQTT the messages are organized in topics, so in order to read the messages from the node we have to be sure to subscribe to the topic in which the Re-mote is publishing, in the default example this topic is *iot-2/evt/fmt/json*.

Connecting to the broker and subscribing to the quoted topic we see the messages coming from the MQTT client in the default Json format:



```
1461768160: Opening ipv4 listen socket on port 1884.
1461768160: Opening ipv6 listen socket on port 1884.
1461768181: New connection from 127.0.0.1 on port 1884.
1461768181: New client connected from 127.0.0.1 as MQTT_FX_Client (c1, k60
1461768181: Sending CONNACK to MQTT_FX_Client (0, 0)
1461768187: Received SUBSCRIBE from MQTT_FX_Client
1461768187:     home/garden/fountain (QoS 0)
1461768187: MQTT_FX_Client 0 home/garden/fountain
1461768187: Sending SUBACK to MQTT_FX_Client
1461768197: Received PUBLISH from MQTT_FX_Client (d0, q0, r0, m0, 'home/ga
1461768197: Sending PUBLISH to MQTT_FX_Client (d0, q0, r0, m0, 'home/garde
1461768257: Received PINGREQ from MQTT_FX_Client
1461768257: Sending PINGRESP to MQTT_FX_Client
1461768317: Received PINGREQ from MQTT_FX_Client
1461768317: Sending PINGRESP to MQTT_FX_Client
1461768377: Received PINGREQ from MQTT_FX_Client
1461768377: Sending PINGRESP to MQTT_FX_Client
1461768412: Received PUBLISH from MQTT_FX_Client (d0, q0, r0, m0, 'home/ga
1461768412: Sending PUBLISH to MQTT_FX_Client (d0, q0, r0, m0, 'home/garde
1461768472: Received PINGREQ from MQTT_FX_Client
1461768472: Sending PINGRESP to MQTT_FX_Client
1461768532: Received PINGREQ from MQTT_FX_Client
1461768532: Sending PINGRESP to MQTT_FX_Client
1461768592: Received PINGREQ from MQTT_FX_Client
1461768592: Sending PINGRESP to MQTT_FX_Client
1461768640: New connection from aaaa::212:4b00:615:a0e6 on port 1884.
1461768652: Received PINGREQ from MQTT_FX_Client
1461768652: Sending PINGRESP to MQTT_FX_Client
1461768662: New client connected from aaaa::212:4b00:615:a0e6 as d:mqtt-de
1461768662: Sending CONNACK to d:mqtt-demo:cc2538:00124b15a0e6 (0, 0)
1461768708: Received SUBSCRIBE from d:mqtt-demo:cc2538:00124b15a0e6
1461768708:     home/garden/fountain (QoS 0)
1461768708: d:mqtt-demo:cc2538:00124b15a0e6 0 home/garden/fountain
1461768708: Sending SUBACK to d:mqtt-demo:cc2538:00124b15a0e6
1461768712: Received PINGREQ from MQTT_FX_Client
1461768712: Sending PINGRESP to MQTT_FX_Client
1461768760: Received PUBLISH from MQTT_FX_Client (d0, q0, r0, m0, 'home/ga
1461768760: Sending PUBLISH to MQTT_FX_Client (d0, q0, r0, m0, 'home/garde
1461768760: Sending PUBLISH to d:mqtt-demo:cc2538:00124b15a0e6 (d0, q0, r0
1461768820: Received PINGREQ from MQTT_FX_Client
1461768820: Sending PINGRESP to MQTT_FX_Client
1461768880: Received PINGREQ from MQTT_FX_Client
1461768880: Sending PINGRESP to MQTT_FX_Client
1461768891: Received PUBLISH from d:mqtt-demo:cc2538:00124b15a0e6 (d0, q0, r0, m0, 'home/garden/fountain', ... (170 bytes))
1461768891: Sending PUBLISH to MQTT_FX_Client (d0, q0, r0, m0, 'home/garden/fountain', ... (170 bytes))
1461768891: Sending PUBLISH to d:mqtt-demo:cc2538:00124b15a0e6 (d0, q0, r0, m0, 'home/garden/fountain', ... (170 bytes))
1461768940: Received PINGREQ from MQTT_FX_Client
1461768940: Sending PINGRESP to MQTT_FX_Client
1461768986: Received PUBLISH from d:mqtt-demo:cc2538:00124b15a0e6 (d0, q0, r0, m0, 'home/garden/fountain', ... (170 bytes))
1461768986: Sending PUBLISH to MQTT_FX_Client (d0, q0, r0, m0, 'home/garden/fountain', ... (170 bytes))
1461768986: Sending PUBLISH to d:mqtt-demo:cc2538:00124b15a0e6 (d0, q0, r0, m0, 'home/garden/fountain', ... (170 bytes))
```

```
{"d":{"myName":"Zolertia RE-Mote platform","Seq #":2,"Uptime (sec)":503,"Def
Route":"fe80::212:4b00:616:fc9","RSSI (dBm)":-93,"On-Chip Temp (mC)":26190,"VDD3
(mV)":3251}}
```

Figure 7: The MQTT.fx application with the node messages and the Mosquitto broker shell in the background

```
{"d":{"myName":"Zolertia RE-Mote platform","Seq
    #":1,"Uptime (sec)":339,"Def
    Route":"fe80::212:4b00:616:fc9","RSSI
    (dBm)":-94,"On-Chip Temp (mC)":26190,"VDD3
    (mV)":3249}}
```

We can see that a lot of information come with the Re-mote messages, analysing the content of the message we can read:

- myName: the hardcoded name of the node
- Seq: the sequential number of the messages
- Uptime (sec): the uptime of the node, in seconds
- Def Route: the route used to deliver the messages
- RSSI (dBm): Received signal strength indication
- On-Chip Temp (mC): the on-Chip temperature
- VDD3 (mV): power

3.2.2 *Connection Performance*

In the previous chapter we described how we managed to implement a working MQTT network, but we did not mention the performance and this is a big issue of this implementation.

Once the node is connected to the border router and to the broker, the network works as expected, but the problem is in the connection phase.

With the default settings the behaviour of the MQTT client in the connection phase is very unpredictable with a time of connection that goes from 10 seconds to some minutes. In certain occasions we have a working connection after 3-4 minutes.

This is clearly an unacceptable frame of time in a scenario as the one we are dealing in this project, since we have to think that during this time we should receive a big amount of data from the sensors. This would create problems both with the data buffering and the data analysis.

Due to of these reasons we decide that it is necessary deeper investigate the connection phase in order to improve the performances into an acceptable range of time.

A detailed evaluation and analysis of the performance are not elaborated because even after few tests it is clear that both the time needed and the random behaviour are not acceptable. This is why instead we directly try to work on them.

3.3 UNDERSTANDING THE NETWORK

After the implementation of the network described in the previous chapter a deepening in the understanding of the network is necessary both to be able to improve the connection performances and to able to properly manage the system looking forward to the future work.

3.3.1 *Need for information*

The first step into this understanding is to have information to analyse in order to see the workflow of the system, as trivial as it seems: without data it is impossible to comprehend what is going on in the platform.

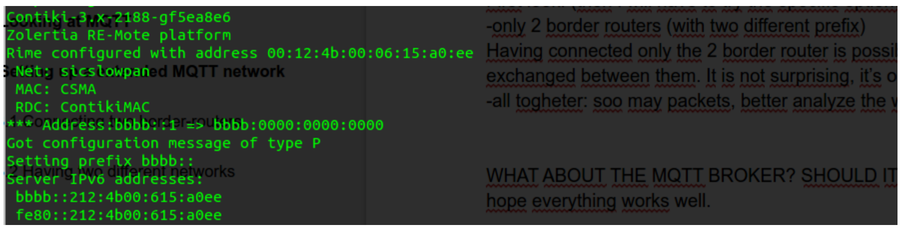
To achieve this goal is necessary to intervene in multiple locations and in different ways.

First of all it is possible to use the *verbosity* option activated both in the mosquitto broker and in the tunslip tool with the parameter *-v*.

Another way to get more data is to enable the debug options in the files involved in the system. This step requires some attention to find out every file used by the system and to build the whole image. IPV6, RPL, sicslowpan, uip, MQTT are some of the protocols involved.

Different files have different ways to activate the debug options, most of them are enabled with the strings *DEBUG PRINT* and *DEBUG 1*.

With these changes the amount of data in our possession increased, we see it comparing the images below:



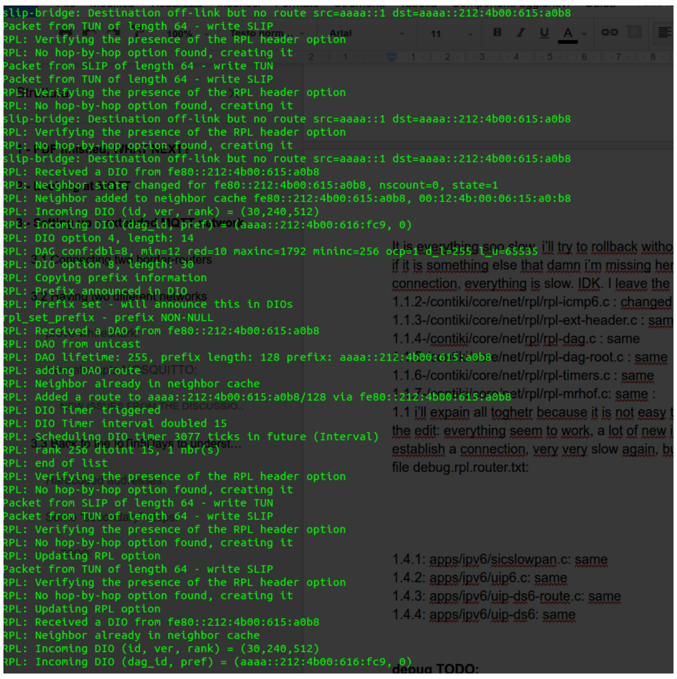
```
Contiki-3.x-2188-gf5ea8e6
Zolertia RE-Mote platform
Rime configured with address 00:12:4b:00:06:15:a0:ee
 Net: sicslowpan
 MAC: CSMA
 RDC: ContikiMAC
*** Address:bbbb::1 => bbbb:0000:0000:0000
Got configuration message of type P
Setting prefix bbbb::
Server IPv6 addresses:
 bbbb::212:4b00:615:a0ee
 fe80::212:4b00:615:a0ee
```

-only 2 border routers (with two different prefix)
Having connected only the 2 border router is possi
exchanged between them. It is not surprising, it's o
-all togheter: soo may packets, better analyze the v

WHAT ABOUT THE MQTT BROKER? SHOULD IT
hope everything works well.

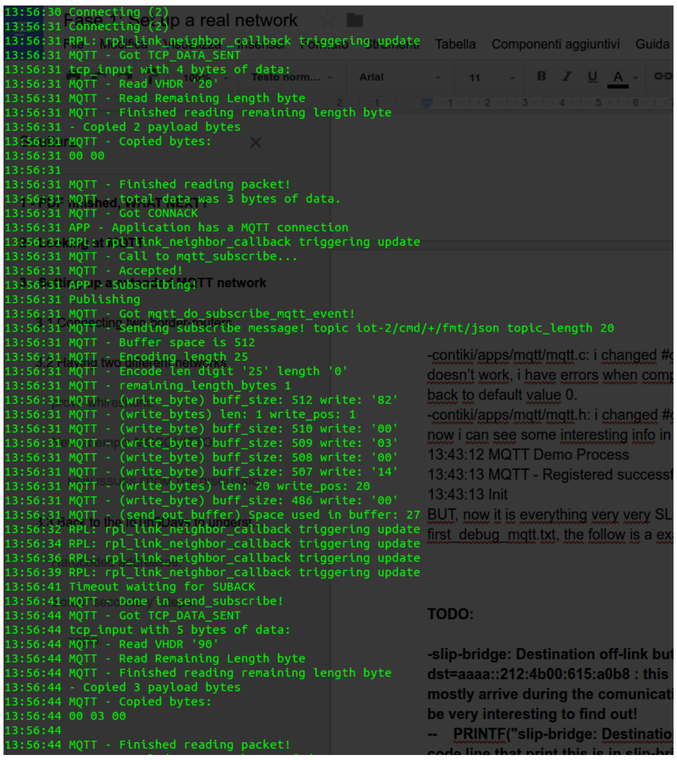
Figure 8: Border router shell before the changes

With this new information is possible to start a deepening into the system and this will involve different sides.



```
slip-bridge: Destination off-link but no route src=aaaa::1 dst=aaaa::212:4b00:615:a0b8
Packet from TUN of length 64 - write SLIP
RPL: Verifying the presence of the RPL header option
RPL: No hop-by-hop option found, creating it
Packet from SLIP of length 64 - write TUN
Packet from TUN of length 64 - write SLIP
RPL: Verifying the presence of the RPL header option
RPL: No hop-by-hop option found, creating it
slip-bridge: Destination off-link but no route src=aaaa::1 dst=aaaa::212:4b00:615:a0b8
RPL: Verifying the presence of the RPL header option
RPL: No hop-by-hop option found, creating it
slip-bridge: Destination off-link but no route src=aaaa::1 dst=aaaa::212:4b00:615:a0b8
RPL: Received a DIO from fe80::212:4b00:615:a0b8
RPL: Neighbor state changed for fe80::212:4b00:615:a0b8, nscount=0, state=1
RPL: Neighbor added to neighbor cache fe80::212:4b00:615:a0b8, 00:12:4b:00:06:15:a0:b8
RPL: Incoming DIO (id, ver, rank) = (30,240,512)
RPL: Incoming DIO (dag_id, pref) = (aaaa::212:4b00:616:fc9, 0)
RPL: DIO option 4, length: 14
RPL: DAG conf:dbl=8, min=12 red=10 maxinc=1792 mininc=256 ocp=1 d_l=255 l_u=65535
RPL: DIO option 8, length: 30
RPL: Copying prefix information
RPL: Prefix announced in DIO
RPL: Prefix set - will announce this in DIOs
rpl_set_prefix - prefix NON-NULL
RPL: Received a DAO from fe80::212:4b00:615:a0b8
RPL: DAO from unicast
RPL: DAO lifetime: 255, prefix length: 128 prefix: aaaa::212:4b00:615:a0b8
RPL: adding DAO route
RPL: Neighbor already in neighbor cache
RPL: Added a route to aaaa::212:4b00:615:a0b8/128 via fe80::212:4b00:615:a0b8
RPL: DIO Timer triggered
RPL: DIO Timer interval doubled 15
RPL: Scheduling DIO timer 3077 ticks in future (Interval)
RPL: rank 256 dioint 15, 1 nbr(s)
RPL: end of list
RPL: Verifying the presence of the RPL header option
RPL: No hop-by-hop option found, creating it
Packet from SLIP of length 64 - write TUN
Packet from TUN of length 64 - write SLIP
RPL: Verifying the presence of the RPL header option
RPL: No hop-by-hop option found, creating it
RPL: Updating RPL option
Packet from TUN of length 64 - write SLIP
RPL: Verifying the presence of the RPL header option
RPL: No hop-by-hop option found, creating it
RPL: Updating RPL option
RPL: Received a DIO from fe80::212:4b00:615:a0b8
RPL: Neighbor already in neighbor cache
RPL: Incoming DIO (id, ver, rank) = (30,240,512)
RPL: Incoming DIO (dag_id, pref) = (aaaa::212:4b00:616:fc9, 0)
```

Figure 9: Border router shell after the changes



```
13:56:30 Connecting (2)
13:56:31 Connecting (2)
13:56:31 RPL: rpl_link_neighbor_callback triggering update
13:56:31 MQTT - Got TCP_DATA_SENT
13:56:31 tcp_input with 4 bytes of data:
13:56:31 MQTT - Read VHDR '20'
13:56:31 MQTT - Read Remaining Length byte
13:56:31 MQTT - Finished reading remaining length byte
13:56:31 - Copied 2 payload bytes
13:56:31 MQTT - Copied bytes:
13:56:31 00 00
13:56:31
13:56:31 MQTT - Finished reading packet!
13:56:31 MQTT - total data was 3 bytes of data.
13:56:31 MQTT - Got CONNACK
13:56:31 APP - Application has a MQTT connection
13:56:31 RPL: rpl_link_neighbor_callback triggering update
13:56:31 MQTT - Call to mqtt_subscribe...
13:56:31 MQTT - Accepted!
13:56:31 APP - Subscribing!
13:56:31 Publishing
13:56:31 MQTT - Got mqtt_do_subscribe_mqtt_event!
13:56:31 MQTT - Sending subscribe message! topic iot-2/cmd/+/fmt/json topic_length 20
13:56:31 MQTT - Buffer space is 512
13:56:31 MQTT - Encoding length 25
13:56:31 MQTT - Encode len digit '25' length '0'
13:56:31 MQTT - remaining_length_bytes 1
13:56:31 MQTT - (write_byte) buff_size: 512 write: '82'
13:56:31 MQTT - (write_bytes) len: 1 write_pos: 1
13:56:31 MQTT - (write_byte) buff_size: 510 write: '00'
13:56:31 MQTT - (write_byte) buff_size: 509 write: '03'
13:56:31 MQTT - (write_byte) buff_size: 508 write: '00'
13:56:31 MQTT - (write_byte) buff_size: 507 write: '14'
13:56:31 MQTT - (write_bytes) len: 20 write_pos: 20
13:56:31 MQTT - (write_byte) buff_size: 486 write: '00'
13:56:31 MQTT - (send_out_buffer) Space used in buffer: 27
13:56:32 RPL: rpl_link_neighbor_callback triggering update
13:56:34 RPL: rpl_link_neighbor_callback triggering update
13:56:36 RPL: rpl_link_neighbor_callback triggering update
13:56:39 RPL: rpl_link_neighbor_callback triggering update
13:56:41 Timeout waiting for SUBACK
13:56:41 MQTT - Done in send_subscribe!
13:56:44 MQTT - Got TCP_DATA_SENT
13:56:44 tcp_input with 5 bytes of data:
13:56:44 MQTT - Read VHDR '90'
13:56:44 MQTT - Read Remaining Length byte
13:56:44 MQTT - Finished reading remaining length byte
13:56:44 - Copied 3 payload bytes
13:56:44 MQTT - Copied bytes:
13:56:44 00 03 00
13:56:44
13:56:44 MQTT - Finished reading packet!
```

Figure 10: MQTT client shell after the changes

3.3.2 *IPv6 Addresses*

IPv6 is one of the most important technologies in this project, a proper comprehension of the addresses characteristics is necessary.

First of all it is important to distinguish between link-local and global addresses:

- **link-local addresses**: These addresses start with the *FE80::/1* prefix (not editable) and are used to communicate with other nodes in the same network.
- **unique local addresses**: addresses that start with the prefix FC00::/7 and are used for local communications inside a network.
- **global addresses**: Equivalent to the IPv4 public addresses, they are unique in the whole Internet and can be used to send a packet to different networks.

In our project the communication is inside one single network, so link-local addresses are used as showed in the MQTT messages from the node.

An important special address is **ff02::1** since IPv6 does not implement broadcast addressing, this is the address used for this purpose. For example at the end of the bootstrap this is the address to whom the node sends a DIS message trying to reach any border router in range.

The node address in contiki can be handled in 2 different ways, it can be hardcoded or not hardcoded. As it is easy to understand, in the **hardcoded** way it is possible to specify the address in the code with the *IEEE ADDR CONF HARDCODED* option. Otherwise in the not hardcoded option, the default one, the address is build from the MAC, so it can be somehow considered static too.

In both the possibilities address prefix is negotiated with the border router and it depends on the network prefix. This behaviour influences both the link-local and the global address.

We tried various configurations about the addresses involving **different prefixes** (from *aaaa::/64* to *fd00::/64*) and hardcoded / not hardcoded settings, but no improvements have been seen in the connection performances.

Because of this we decided to use the default option, with not hardcoded addresses build up from the MAC.

3.3.3 *Identifying the problem*

Since the variables involved in the connection problem are several we decide to make a test to get closer to the issue.

The test consists in leave a part MQTT and go back to the udp-client example. In this way we can understand if the long time needed to the node to connect to the border router is due to of MQTT or not.

With the udp example we have the same performances and the same behaviour in the connection phase. Despite this could seem a bad news, it actually is not because tells us that MQTT is not the problem.

The investigation consists in the analysis of the border router and MQTT-client logs and after a careful deepening we have been able to identify the problem comparing the usual working flow of RPL [18] and our logs.

Looking into the RPL protocol:

As expected at the end of the bootstrap phase, in our MQTT client example this means at the end of the Init phase, we see the node correctly broadcast a DIS message:

```
RPL: Sending a DIS to ff02::1a
```

This message is properly received from our border router and it answers in the right way to it with a DIO message:

```
RPL: Received a DIS from fe80::11:22ff:fe33:4401
RPL: Multicast DIS => reset DIO timer
RPL: Sending a multicast-DIO with rank 256
```

So far everything works properly according to the RPL standard, but after this message the problem takes place.

This DIO is not properly received by the node, it takes a long and, most worrying, random time to be received.

Most of the times we can see this cycle repeating several times with the node sending some DISs and the border router always answering with a DIO, before finally it gets caught by the node, processed and the connection established as it possible to see in the log messages below:

```
RPL: Received a DIO from fe80::212:4b00:616:fc9
RPL: Neighbor state changed for fe80::212:4b00:616:fc9,
    nscount=0, state=1
RPL: Neighbor added to neighbor cache
    fe80::212:4b00:616:fc9
RPL: Incoming DIO (id, ver, rank) = (30,240,256)
RPL: Incoming DIO (dag_id, pref) =
    (fd00::212:4b00:616:fc9, 0)
RPL: DIO option 4, length: 14
RPL: DAG conf:dbl=8, min=12 red=10 maxinc=1792 mininc=256
    ocp=1 d_l=255 l_u=65535
RPL: DIO option 8, length: 30
RPL: Copying prefix information
RPL: New instance detected (ID=30): Joining...
RPL: rpl_add_parent lladdr #0x20004fe8
    fe80::212:4b00:616:fc9
RPL: Adding fe80::212:4b00:616:fc9 as a parent: succeeded
RPL: adding global IP address fd00::212:4b00:616:f3a
RPL: rpl_set_preferred_parent fe80::212:4b00:616:fc9 used
    to be NULL
RPL: Joined DAG with instance ID 30, rank 768, DAG ID
    fd00::212:4b00:616:fc9
```

The problem can be synthesized and easily understood looking at the following image:

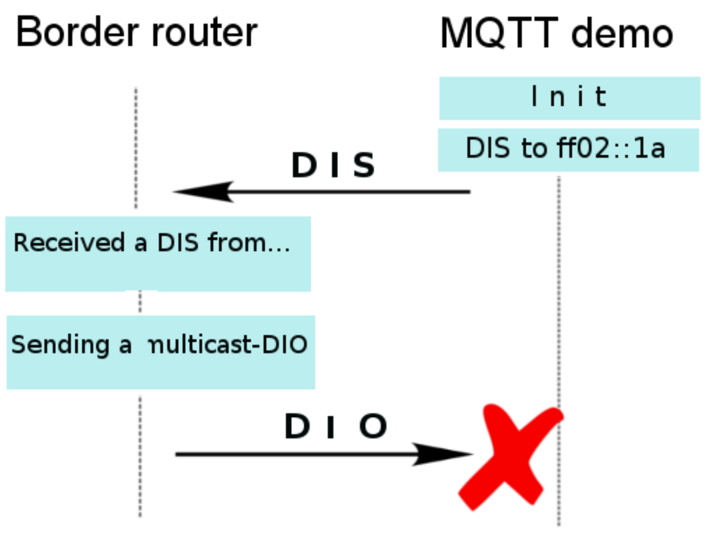


Figure 11: Connection phase problem

After this analysis we finally identified the problem, that takes place in the RPL connection. This is an important step forward and in the next chapter we illustrated how is possible to solve it.

3.4 THE NEW NETWORK

In this chapter we describe how we manage to solve the identified problem and we discuss the performances of the new implemented network, the results of this first part of the project.

3.4.1 *Solving the problem*

In the previous chapter we managed to isolate the problem, but the road to solve it is still not easy to spot. Despite we now know the problem takes place in the RPL protocol, we only know that that is the symptom, but the problem could even be caused by something else.

3.4.1.1 *RPL Rank*

The RPL Rank is one of the suspected for this behaviour, in fact in RPL protocol specifications is possible to see that a node ignores a DIO message from nodes with higher rank than his own.

This routine is necessary to avoid loops in the parent-child network structure.

We already know from the DIO messages sent by the border router that it has a default rank of 256:

```
RPL: Sending a multicast-DIO with rank 256
```

So if the node at the beginning starts with a lower rank it will consequently ignore the DIO.

The rank is not fixed during the life cycle of the nodes and the behaviour observed could be explained by the progressive rising of the node rank while it is searching for a acceptable network until it is higher than 256.

The analysis of the logs shows that the node use to start with a default network of 512 though, as we can see in the lines below, and it changes during the life time but it has never observed to go under the 256 level.

```
RPL: rank 512 dioint 13, 1 nbr(s)
RPL: nbr 238  256,   256 =>  512 d* (last tx 0 min ago)
```

This means that the rank parameter is not what causes our problem.

3.4.1.2 *Radio Duty Cycle*

The Radio Duty Cycle is the next part we look into, because despite the problem we observed takes place in the RPL connection, this could be caused by something else.

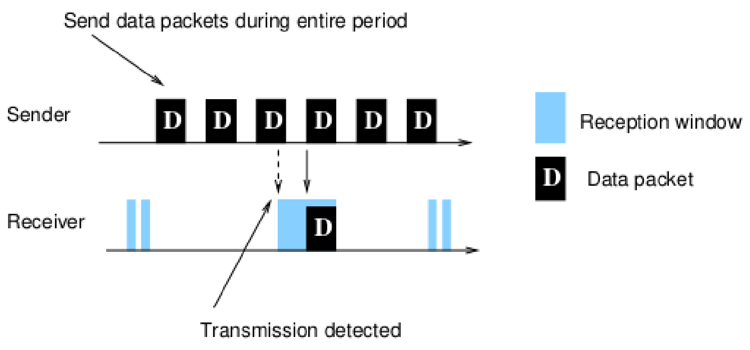


Figure 12: Radio Duty Cycle

In contiki we have available 2 different options to handle the RDC, ContikiMAC and NullRDC[19]. As the names suggest the RDC is not the only thing involved in this choice, but even the MAC (Medium Access Control) layer.

Net: sicslowpan	Net: sicslowpan
MAC: CSMA	MAC: nullmac
RDC: contikimac	RDC: nullrdc

Table 1: ContikiMAC vs NullRDC

ContikiMAC is the default option and it is a very power efficient RDC protocol and it is used with the CSMA MAC layer protocol. Due to its power efficiency, with ContikiMAC the node transceiver is in sleeping mode for most of the time and this is the reason of its power efficiency.

NullRDC is the other RDC mechanism offered by Contiki and it has opposite properties: it lets the node transceiver always up not caring about the power efficiency and it uses the nullmac mechanism instead of CSMA (Carrier Sense Multiple Access) that does not do any MAC layer processing, as it possible to understand from the name.

This is something that can be at the origin of our problem, because the ContikiMAC could be too much extreme in the power efficiency side and so create issues in the packets reception.

The two option are very different one from the other and switch to the use of NullRDC is a drastic change, considering the importance of the power efficiency in the scenario of this project.

So before to take this important choice it is worth to try to improve ContikiMAC instead directly go on trying NullRDC.

This can be done with the *CONTIKIMAC CONF WITH PHASE OPTIMIZATION*. As the name suggests this option enable/disable the phase optimization and it is disabled by default because it is not available for all the platforms.

It is not clear if the RE-mote platform is supported, but we can observe that no improvements are shown with the activation of this option.

Consequently the only option remained is to switch from ContikiMAC to NullRDC and this is done by specifying the new options in the *project-conf.h* files of both the border router and the mqtt-demo examples. In this way we dont change the default settings of the whole platform.

```
//specify what RDC driver Contiki should use
#define NTSTACK_CONF_RDC nullrdc_driver || contikimac
//specify what MAC driver Contiki should use
#define NETSTACK_CONF_MAC nullmac_driver || csma
```

These new settings clearly show a different and better behaviour in the connection phase of the node, that is described in the next sub chapter.

3.4.2 *Results*

This new settings bring with them big differences in the network.

In fact in the IoT scenario where we have to handle low resources devices the power efficiency is very important and the choice of NullRDC instead of ContikiMAC of course affect that side.

In addition, we are renouncing at the MAC layer too, considering that mixed approaches, based on NullRDC and CSMA or using the NullRDC settings only in the client node, are not supported.

These choices have big consequences (that we will discuss further in the work), but looking at the good results that we are going to show and considering that the main goal of this project is to be focused on the Handover, it seems to be the best choice to go on in the development of this work in this way.

The Results

These data are taken from the final project tests. The DIS to DIO time is a little higher than the real first connection time because in the handover the node discards the DIO messages from the old network waiting for the new ones.

The meaning of these data are just to show how the random behaviour of the connection has been stabilized and the average time for a connection is passed from minutes to seconds.

An acceptable value that will be further investigated in the next chapters and for the moment is enough for us to continue the work.

Average DIS to DIO	430.87	157.887
Average DIO to RPL reconnect	5.09	0.471

Table 2: Results in ticks

In the last column is showed the standard deviation.

This give as an *Average time from DIS to DIO* of 3,360786 seconds and an *Average time from DIO to RPL reconnect* of 0,039702 seconds. At this time it has to be added the average time of the MQTT connection that is 6,138678s .

Average DIS to DIO	3,360786 seconds
Average DIO to RPL reconnect	0,039702 seconds

Table 3: Results in seconds

The main point we want to focus on is that after these changes we have a stable connection phase that needs less than 10 seconds, against the previous random behaviour that certain times needed 3 or 4 minutes to establish a connection.

4 DEFAULT HANDOVER IMPLEMENTATION

The handover is the main argument discussed in this project, after we set up a working network in the chapter 3, in this chapter we analyse how it works in the default implementation of the system.

4.1 HANDOVER BACKGROUND

First of all it is important to clearly discuss what an handover is and why it is so important that is the core of this work. Then in this sub chapter we show the work necessary to get the platform ready to perform it.

4.1.1 *The handover definition*

A handover is a change of network the node is connected to.

The reasons that leads to this change can be of various forms and strongly dependent on the user case scenario. The easiest way to see it is to imagine that the node is not any more under the cover range of its border router.

This can happen both if the node is changing its position or if the first border router is not reachable any more for any reason.

To be out of cover or observing the signal strength are just basic ways to see the need of an handover, there is a big amount of studies on this issue and the answer is strongly dependent on the scenario.

A handover could be useful to have a better work balance in a system or it could be useful to improve the power efficiency changing to a border router with more battery for example. A handover could even be triggered by the application layer with

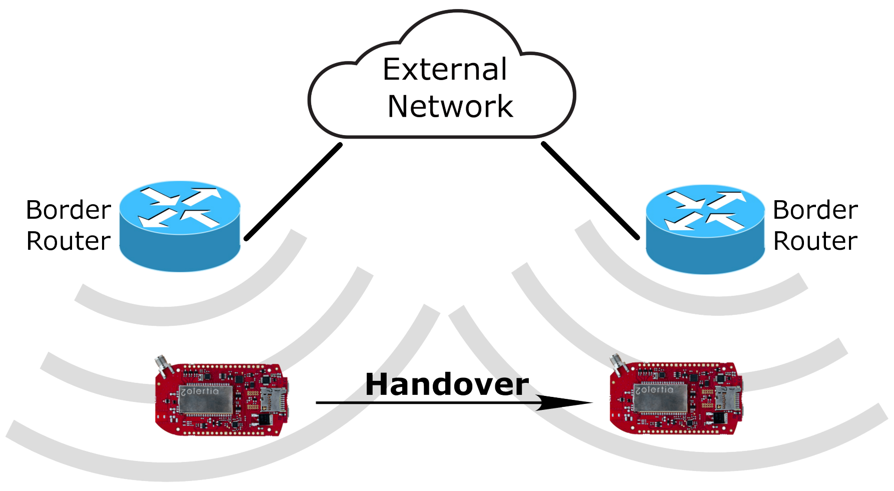


Figure 13: Handover illustration

the proper structure.

Other than the reason that brings us to an handover, very important is even to understand what happens during a handover in order to properly handle it. This is important because the handover is a heavy operation for a low-resource system, we have to keep in mind that this project is about low-resource devices and the transmission phase is one of the most demanding.

In the medical area scenario for example a properly handled handover could let patients being easily moved from one area to another without any worrying about the extremely important and various data the sensor are controlling.

In order to do that we need to know how an handover takes places, what happens in the device, what changes are shown and after that in case try to improve the standard behaviour in our purposes view.

4.1.2 *Setting up two different networks*

In order to perform a handover it is of course necessary to have at least two different working border routers, in the purpose of this project we are mostly interested in have two complete different networks (i.e. different prefixes) to better simulate a real scenario and this is what we are going to implement.

Theoretically a handover can be performed even inside the same network going from one border router to another, but in the Contiki RPL implementation we have only one active border router in charge at a time, another border router will become active only if the first one is not available for any reason.

As we said in chapter 3.1.2 to connect a router we use the command

```
make connect-router
```

that starts the border router activities and connects the node to the local machine through the Tunslip utility.

It is interesting to note that since we need to start the border router with this command, we need to have them connected to our host by an USB cable and this is why we cant have a battery powered border router in the scenario of this project.

To implement 2 different networks we need to make some change to this routine. We need to create 2 different directories because we need to edit the Makefile of each border router.

The prefixes are defined in the Makefile with the following lines:

```
ifeq ($(PREFIX),)
 PREFIX = fd00::1/64
endif
```

```
ifeq ($(PREFIX),)
 PREFIX = fd22::1/64
endif
```

As it possible to see the prefix of the first network is *fd00::/64*, as advised in the Contiki guide, and the second prefix is *fd22::/64*, that is arbitrary.

Since we have both the border router connected to our host, we need to edit the call to Tunslip too, because we need to create 2 different bridges to the local machine, this is done by adding the *-s [siodev]* parameter.

First router:

```
connect-router: $(CONTIKI)/tools/tunslip6
   sudo $(CONTIKI)/tools/tunslip6 -L -v3 $(PREFIX)
```

Second router:

```
connect-router: $(CONTIKI)/tools/tunslip6
   sudo $(CONTIKI)/tools/tunslip6 -s ttyUSB1 -L -v3
       $(PREFIX)
```

The last change needed is in the command used to start the border router, since now we have 2 of them connected at the same time we need to specify the device with the *port* parameter:

```
make PORT=/dev/ttyUSB0 connect-router
```

With these changes we are able to connect the 2 border routers and have 2 different networks. As it is possible to see in the border router shells they create 2 different DODAGs:

First router:

```
RPL: Node set to be a DAG root with DAG ID
    fd00::212:4b00:616:fc9
#A root=201
```

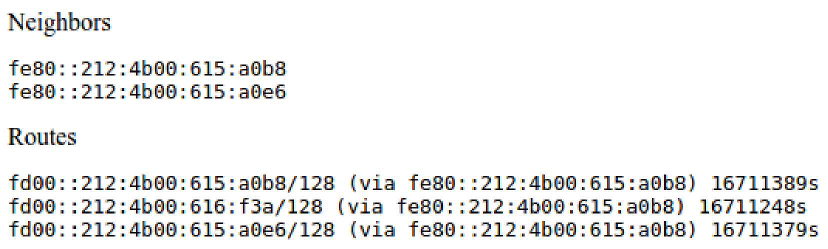
Second router:

```
RPL: Node set to be a DAG root with DAG ID
    fd00::212:4b00:615:a0ee
#A root=238
```

Another helpful tool to see the networks situation is the built-in webserver in the border router example. This is a simple web page reachable inserting the border router IPv6 address in the browser (into brackets) that shows the neighbours and the established routes of the border router, like in this image:

4.2 PERFORMANCE EVALUATION

In this chapter we describe the tests done on the handover and the implications that the results show.


Neighbors

```
fe80::212:4b00:615:a0b8
fe80::212:4b00:615:a0e6
```

Routes

```
fd00::212:4b00:615:a0b8/128 (via fe80::212:4b00:615:a0b8) 16711389s
fd00::212:4b00:616:f3a/128 (via fe80::212:4b00:615:a0b8) 16711248s
fd00::212:4b00:615:a0e6/128 (via fe80::212:4b00:615:a0b8) 16711379s
```

Figure 14: Border router webserver example

Before we show how we performed the test is important to talk about the **cover range** of the Re-mote border router, because to force the handover we need to move the node out of the range of its first border router and into the range of the second one.

The Zolertia Re-mote is equipped with the CC2538 2.4-GHz transceiver by Texas Instruments [20]. The cover range changes a sensibly considering the use of an antenna as it is easy to imagine.

We have a range about 30 meters with the use of the antenna, but under a meter if we use it without.

Since we proved that the use of the antenna has implications only in the cover range and not in the correct transmission of the packets, we removed the antenna for the tests in order to have little ranges in which we have to move our node to force the handover.

4.2.1 *Test description*

The first part of the test consists in defining the positions of the 2 border routers in such a way that we have an overlapped area of the 2 networks as shown in the image below:

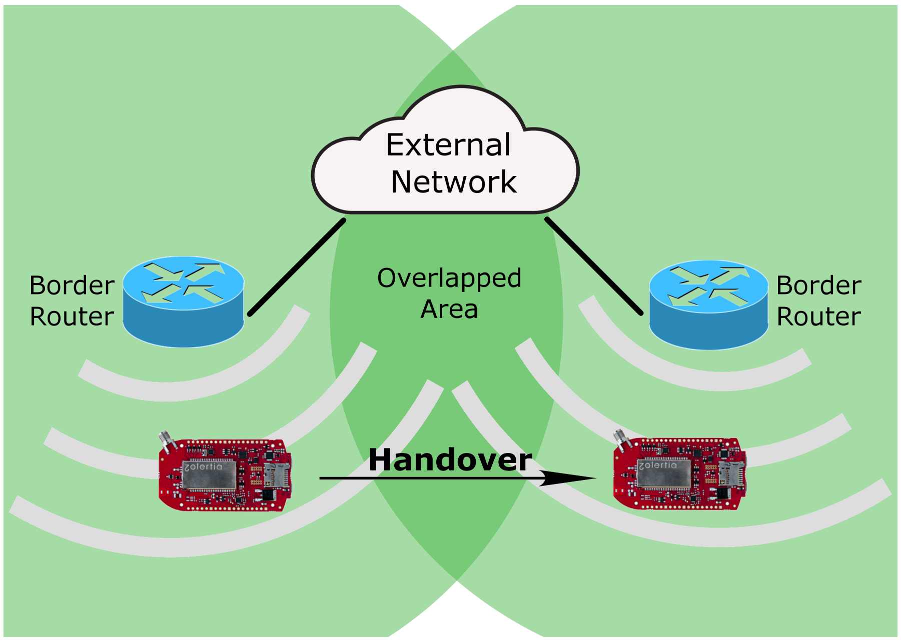


Figure 15: Networks cover range during tests

We define 3 static positions, by empiric tests, in which the node is moved step by step. The positions are chosen with this purpose:

- Position 1: covered only by network 1 (P1)
- Position 2: covered by both the networks (P2)
- Position 3: covered by network 2 (P3)

With the resulting configuration:
The role of Position 2 is important to understand what happen when the node is under the cover range of both the networks, otherwise to only perform the handover 2 positions could be enough.

The tests are done by 4 steps:

1. We start the node in P1
2. After the node initialization is done we move to P2
3. After an arbitrary time of 30 seconds we move on P3
4. Behaviour observation

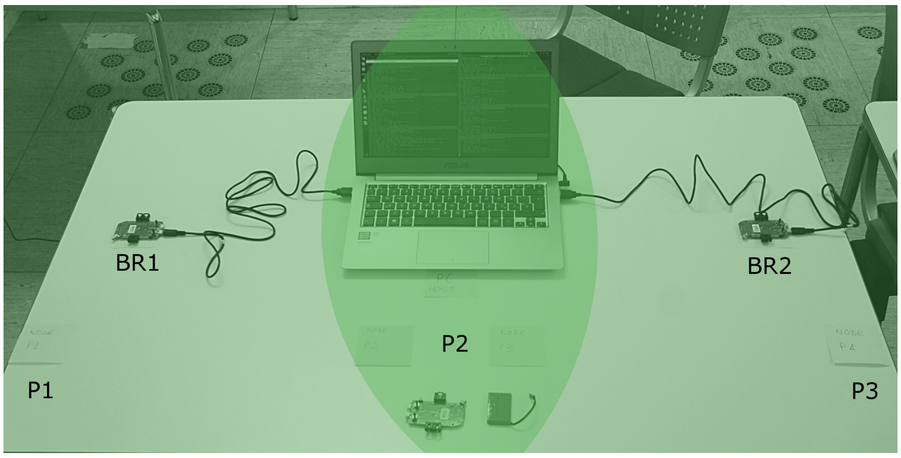


Figure 16: Tests configuration

4.2.2 *Results*

First of all we describe the observed node behaviour in the first test following the step by step procedure to understand what goes on in the system. We use the border router webservers to look into the events.

4.2.2.1 *First test*

Step 1: the mote is started in Position 1 and it establishes connection with the border router 1, the border router 2 is out of range and so it does not show anything about the node in its webserver:

First router:

```
Neighbours
fe80::212:4b00:615:a0ee
fe80::212:4b00:615:a0e6
Routes
fd00::212:4b00:615:a0e6/128 (via fe80::212:4b00:615:a0e6)
```

Second router:

```
Neighbours
fe80::212:4b00:616:fc9
Routes
```

Step 2: the node is moved to Position 2, it enters the cover range of border router 2 and looking at its webserver we see that it has been added as a neighbour. Despite that no routes are established between the node and the border router 2:

First router:

```
Neighbours
fe80::212:4b00:615:a0ee
fe80::212:4b00:615:a0e6
Routes
fd00::212:4b00:615:a0e6/128 (via fe80::212:4b00:615:a0e6)
```

Second router:

```
Neighbours
fe80::212:4b00:616:fc9
fe80::212:4b00:615:a0e6
Routes
```

Step 3: the node is moved to Position 3, it goes out of the border router 1 cover range and now it is only under the border router 2 one. At this point we have all the conditions that should induce the handover, but instead we are not able to see anything happen in the first place.

Only after some minutes we are able to see the node led to start middle-fast blinking telling us that the node was in a connection phase. After that we are able to see that the mote establish a new connection with the border router 2 as is shown both in the border router 2 webserver with the new routes and in the broker shell with a new connection:

First router:

```
Neighbours
fe80::212:4b00:615:a0ee
fe80::212:4b00:615:a0e6
Routes
fd00::212:4b00:615:a0e6/128 (via fe80::212:4b00:615:a0e6)
```

Second router:

```
Neighbours
fe80::212:4b00:616:fc9
fe80::212:4b00:615:a0e6
Routes
fd22::212:4b00:615:a0e6/128 (via fe80::212:4b00:615:a0e6)
```

Note: the route between the node and the border router 1 is still visible because it has a lifetime defined in the code before it expires.

This first test clearly shows a problem in the handover behaviour because the time needed to complete the operation is about minutes when we necessarily need a time much smaller than that to be possible consider to use it in a real scenario.

One test is of course not enough to evaluate the performance of a system and we went on with the experiments, but after 20 of them behaviour was not acceptable and improvements were needed.

In fact after 20 tests we register the following results to complete the handover (i.e. to establish a new connection):

Average Handover time	163,25 seconds	2′ 43″
Standard deviation	149.08 seconds	

Table 4: Standard handover results

These measurements are done looking at the border router shells that shows the time the messages are received by the uptime of the board in seconds.

Someone could object that this is not a measurement precise enough, and this could be true but looking at the magnitude of the measurements they are clear enough to require changes. This is the reason why no other measurements are considered needed.

Very important instead is to understand why the system needs so much time to perform a handover and what happen during that time, this is what we do in the next chapter.

4.3 ANALYSIS AND CONSIDERATIONS

After the default handover test the first thing we want to do is to investigate the behaviour of the node during the handover to understand what makes it so long and where we can focus on to improve the performances.

Analysing the log files we find that the longest phases involved in the handover, when we move the node from Position 2 to Position 3 out of its network, are 2:

- MQTT disconnection from the broker
- RPL reconnection to the new network

MQTT DISCONNECTION When the node is out of its network it has not any event that makes it reacts, so when the Publishing event timer is triggered the node tries to publish the message to the Broker.

But being out of range this is impossible and so the node is stuck in the Publishing phase for a long time, in the node shell is possible to see in fact a long series of Publishing messages that are printed during that phase.

This phase ends when the broker disconnects the node because it is not receiving any answer to is PINGREQ messages.

RPL RECONNECTION The second problem is about the RPL reconnection, in fact while the node enters new networks it starts to receive the DIO messages from the new border router, messages sent by the border router because of its DIO even timer.

The problem is that the node in the first place ignores those messages because they are from a different DAG, as it possible to see in the following messages from the node shell:

```
RPL: Received a DIO from fe80::212:4b00:615:a0b8
[...]
RPL: Root ignored DIO for different DAG
```

Only when the disconnection from the first network is completed will the node at the end accept the DIO message from the new border router, processes it and establish a new connection.

4.4 CONSIDERATIONS

The time needed to complete a handover is clearly not acceptable in the vision of this project, we can not use minutes to process this operation. We have to remember that we are managing critical patients data.

With the information we found out about the standard handover it is clear that the implementation of Contiki is not thought to involve handovers, the events and triggers necessary for this operation have not sufficient performance.

These are the reasons why instead of going deeper in the analysis of the standard handover we directly decide to implement new mechanisms to handle this operation.

These activities are illustrated in the next chapter.

5 PROPOSED SOLUTIONS

The previous chapter shows the standard implementation of the handover mechanism does not work for our purpose, in a medical area scenario it is not acceptable to use minutes to handle an handover, we need to improve the system.

In order to do that we decided to change our point of view: we are not interested in the handover by movement because we want to be able to trigger the handover ourselves.

We made this decision because it is not the main scope of this project to define when to do a handover, this is a different question that needs further studies to be answered. In fact, to decide when perform a handover is a question that involves possibly a huge amount of variables and most of all it could be strictly dependent on the scenario in which the handover takes place.

Instead we want to focus on how to perform a handover with the best performances possible with our platform.

In order to reach this purpose the first goal of this chapter is to define a way to trigger the handover on our own, manually with a button pressure. This will be the first proposal presented in this thesis.

The second proposal makes a step forward moving the handover activation from a manual activity to an application layer decision. In the future smarter algorithms may be able to handle most of the decisions.

Both the proposals are based on the same handover handler mechanism, that gives them the same performances that will be illustrated in the following chapter.

5.1 MANAGING THE HANDOVER

5.1.1 *Guidelines*

The work to define a new way to perform the handover starts from the results of the standard implementation we illustrated in chapter 4. With those tests we identified the main critical phases in the MQTT disconnection and in the following RPL reconnection, and those are the areas in which we concentrate our efforts to improve the performance.

Since the handover operation involves a large number of subjects, from MQTT to RPL, and most of them are deeply inserted in the platform functioning we have to pay attention at our work. One of the guidelines used to carry on the work is to not twist too much the working flow, but instead use as much as possible the present structure in order to not compromise the stability of the system.

This way of working makes harder at the beginning to edit the code and the system, but at the end it has a good trade-off preserving the system and letting us reuse most of the routines and events in it.

Following these guidelines we define the idea of the handover handler we want to create: using the functions provided by MQTT we request the disconnection from the Mosquitto Broker, after that we disconnect from the RPL network.

Properly handling these actions let us be ready to establish a new connection, but instead of taking care on our own of those connection activities we will use as much as possible the built in connection system.

As it possible to see in the image we handle the descending part through the disconnection from the old network and, with the appropriate adaptations that solve the RPL reconnection issue, we leave the system taking care about the new connection.

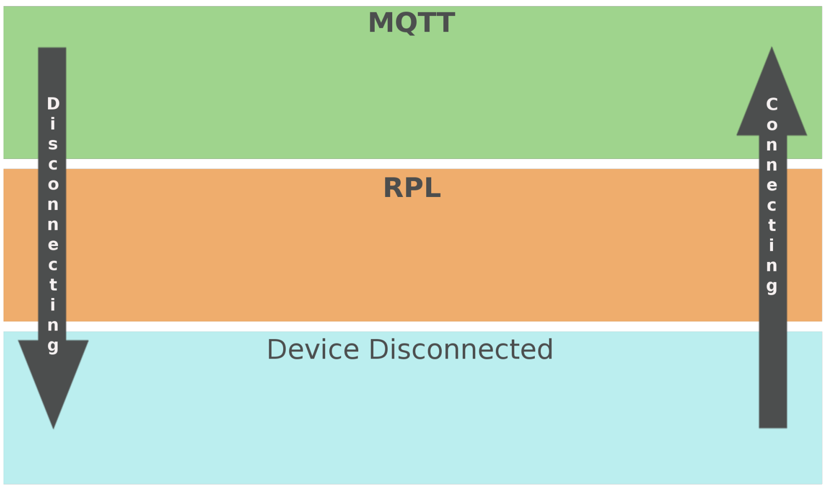


Figure 17: Handover idea

5.1.2 *The implementation*

The first step in the implementation is to find where the disconnection and connection phases are handled, with the work done in the chapter 4, we found that the files involved are:

- *mqtt-demo.c* for MQTT (*contiki/examples/c2538-common/mqtt-demo*)
- *rpl-icmp6.c* for RPL (*contiki/core/net/rpl/*)

In the explanation of the changes done we follow the logic in which the functions are called during the workflow, it is important to note that in the snapped code we present, we cut off the debug information and comments to have a code as clean as possible.

1. **my_handover_handler()** - *mqtt-demo.c*

This custom function we define is the entry point of the entire handover process, this function is called by the trigger events we will show in the next sub chapters.

The role of this function is to start the **MQTT disconnection** from the broker, this is done with the following code:

```
static void
my_handover_handler(void){

   my_disconnected_event=true;
   mqtt_disconnect(&conn);

}
```

my_disconnected_event is a custom boolean variable we use as a flag to say to the system when the custom handover mode is active.

We use this solution because in this way we preserve the standard functioning of the system when we are not interested on performing a handover, this let us leave unaltered the stability of the platform during its normal way of working.

mqtt_disconnect() is a MQTT function that disconnects the client from the broker, using the *conn* parameter, a *struct* that contains all the information about the connection currently used.

2. case STATE_DISCONNECTED - *mqtt-demo.c*

After the MQTT disconnection is done, the *MQTT EVENT DISCONNECTED* event is triggered by the system, this makes the call to the function *state_machine()* with the system state on *STATE DISCONNECTED*.

Entering the relative switch case in which the disconnection is handled:

```
case STATE_DISCONNECTED:
[...]
if(my_disconnected_event==true){
   my_disconnected_event=false;
   start_myhandover();
}
[...]
```

In the snipped code we show the changes done in the default function: we add an *if* condition that checks if the the custom handover mode is active or not, if it is we call the custom function *start_myhandover()* defined in the *rpl-icmp6.c* file.

This is an example about how we tried to use as much as possible to working structure of the system. We start on our own the MQTT disconnection on point 1, but we use the platform events to detect the completion of the procedure.

This is important because the disconnection takes time, we will see it in the results, and before continuing with the handover working-flow we need to wait until this is completed. Otherwise it occurs that we would call the RPL disconnection before the MQTT disconnection is completed facing a failure in the system.

The disconnection is confirmed by a led that starts blinking and by the broker shell in which we can see the following messages:

```
Received DISCONNECT from d:mqtt-demo:cc2538:00124b160f3a
Client d:mqtt-demo:cc2538:00124b160f3a disconnected.
```

3. **start_my_handover()** - *rpl-icmp6.c*

Calling this function we pass to the *rpl-icmp6.c* file, this says us that we are in the RPL protocol area. In fact this custom function handles the RPL disconnection, one of the step that needed more work on it, despite the final easy looking code.

```
void
start_myhandover(void)
{
   mydag = rpl_get_any_dag();

   rpl_free_instance(mydag->instance);

   dis_output(NULL);
}
```

rpl_get_any_dag() is a RPL built-in function that retrieves the current dag struct and save it to the custom *mydag* variable. This variable is important because lets us remember the old network, otherwise after the disconnection we would not know which was the last network we were connected to.

rpl_free_instance() is a RPL built-in function too, it manages the disconnection from the current RPL instance, that we pass as a parameter using *mydag->istance*.

This function is defined in the *rpl-dag.c* file (*contiki/core/net/r-pl/*) and it is important to look at its code to understand how the RPL disconnection is handled:

```
void
rpl_free_instance(rpl_instance_t *instance)
{
[...]
  for(dag = &instance->dag_table[0], end = dag +
      RPL_MAX_DAG_PER_INSTANCE; dag < end; ++dag) {
    if(dag->used) {
      rpl_free_dag(dag);
    }
  }
[...]
}
```

It is possible to see that the main activity of this code is to call another RPL built-in function, *rpl_free_dag(dag)*. This function, in order:

1. sets the DAG as unused
2. removes the routes
3. removes the parent

dis_output() is the last RPL built-in function called.

After *rpl_free_instance()* is processed our node is completely disconnected from the network and so it is ready to start a new connection routine.

As we explained in chapter 2, the RPL connection starts with the broadcasting of a DIS message from the node and this is the function that let us do that. The *NULL* parameter corresponds to a broadcasted DIS, instead of a unicast message if we would pass an address as a parameter. After the disconnection phase from MQTT and from RPL, with this function we send a new

DIS message from the node and enter again the standard connection routine.

4. dio_input() - *rpl-icmp6.c*
As we said in the 5.1.1 Guidelines, to stay close to the usual system workflow costed us efforts in the complete understanding of the platform, but it has its good trade-off, like in this case.

dio_input() is the function activated by the system any time the node receives a DIO message. As we saw in chapter 2 the RPL connection mechanism is started when the border router receives the node DIS and answers with a DIO message, here again the same image as a reminder:

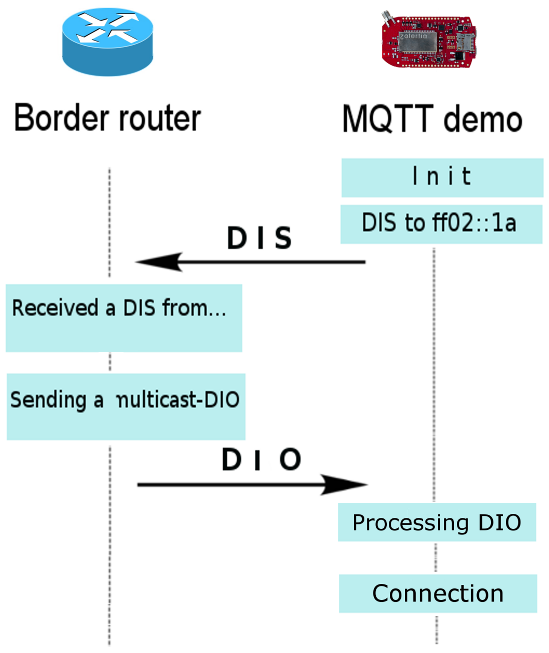


Figure 18: The RPL connection phase

What in this image is in the block Processing DIO is done, in part, in the *dio_input()* function.

But we had to add something to this code in order to work out one major aspect of a Handover: every time we perform a handover we want to change network and in order to achieve this goal we need to create a filter in this function that avoid the processing of the old border router DIO messages.

This is done adding if conditions in the code of the *dio_input()* function.

```
static void
dio_input(void)
{
[...]
   if(mydiocheck==true){
      if(uip_ipaddr_cmp(&dio.dag_id,&mydagid)){
         goto discard;
      }else{
         //do nothing
      }
   }
[...]
  rpl_process_dio(&from, &dio);

 discard:
  uip_clear_buf();
}
```

The first *if* condition checks the value of the mydiocheck variable, a boolean variable we use to activate/deactivate the custom handover mode. In this way we are able to preserve the usual behaviour of the system when we are not performing any handover.

In the second one instead, we do the actual comparison between the old network we were connected to and the new DIO message arrived. This is done using the DODAG ID as parameter of the comparison:

```
 uip_ipaddr_cmp(&dio.dag_id,&mydagid)
```

At this point the node is not connected to any DODAG, but in point 3 we saved the previous DODAG ID in the *mydagid* variable.

- If the condition is **true** it means the DIO we received is from the old border router, so we can not accept it in order to perform the handover and change to a new network. This is done by the built-in function *uip_clear_buf()*

that will delete any information about the received DIO cleaning the buffer.

- If the condition is **false** we have received a DIO from a new network: this means that the *dio_input()* has elaborated the DIO message and can now call the function that will finalize the processing of the DIO message and establish the new RPL connection: *rpl_process_dio()* .

The following diagram summarize the working flow of the filter we add at the *dio_input()* function:

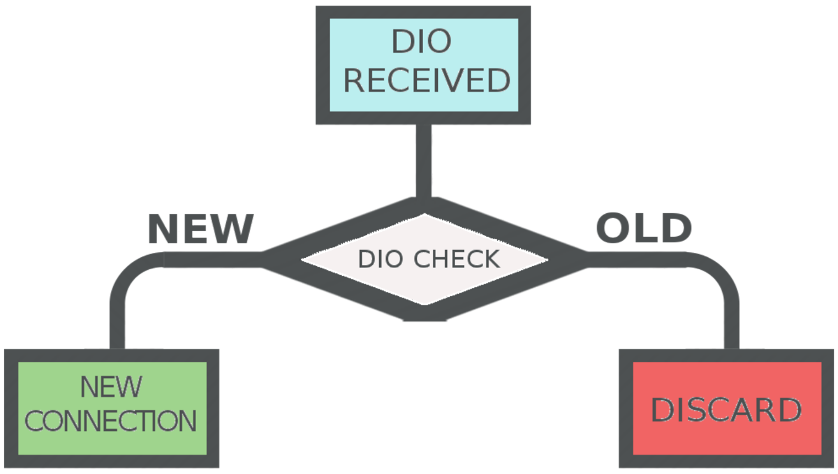


Figure 19: DIO filter workflow

5. New connection
After the DIO processing is done in the *rpl_process_dio()* function the system establishes a new connection to the new RPL DODAG and then to the broker, as we can see in the following messages:

```
New connection from fd22::212:4b00:616:f3a on port 1884.
New client connected from fd22::212:4b00:616:f3a as
    d:mqtt-demo:cc2538:00124b160f3a (c1, k11520,
    u'use-token-auth').
```

It is important to note that the address of the MQTT broker is coded in the node, we assume that this is static and the node knows that. In this way even if the node is connected to a new network, with a different prefix, it is able anyway to connect to its broker and deliver properly the MQTT messages.

At the end of the process we are now able to provide a more detailed image of the whole operation:

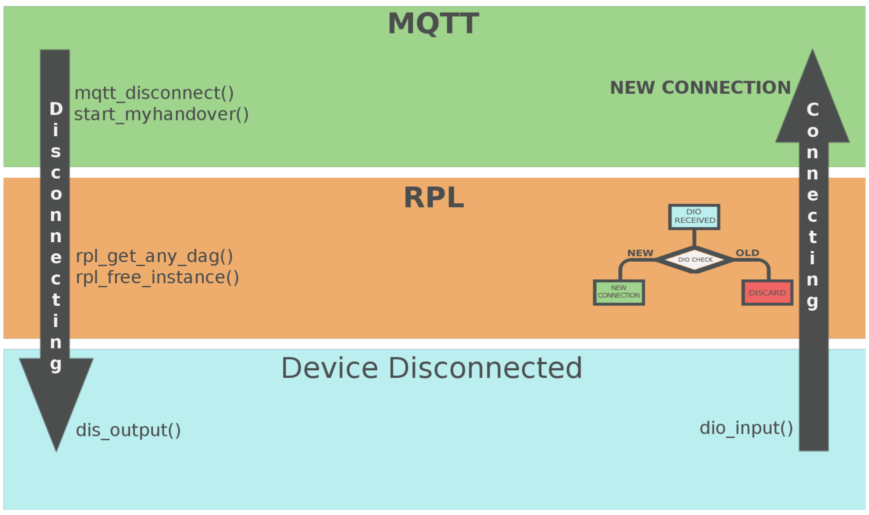


Figure 20: Handover operations

This custom handover mechanism we defined is used in both the proposals we show in the next chapter.

5.2 PROPOSALS

Over the handover mechanism presented in the previous chapter we built 2 different proposals. The main practical difference between these proposals is how we trigger the handover, but this change implies a big logic difference indeed because we move from a manual triggered handover to a handover triggered by the application layer.

Even if in the redaction of this thesis the proposals and the custom handover implementation are presented separately, they were developed side by side and the two parts should be considered as one. This choice aims to improve the readability of the thesis.

The logic behind the definition of the proposals is to improve the 2 biggest issues identified in the standard handover in chapter 4, the MQTT disconnecting and the RPL connecting phases.

5.2.1 *Manual handover*

This first proposal provides a manual trigger to start the handover operation, the trigger is represented by the user button available in the Re-mote board.

With minor changes in the *mqtt-demo.c* file we are able to programme the button to start the handover calling the custom function *my_handover_handler()*:

```
if(ev == sensors_event && data == PUBLISH_TRIGGER){
   my_handover_handler();
}
```

This is the *if* condition we inserted in the MQTT *PROCESS THREAD*, that basically calls the *my_handover_handler()* function, presented in the previous chapter, that starts the handover when the user button is pressed.

This is a raw implementation of the handover, started by a manual event that implies a direct human interaction.

Despite that this proposal is a first way we are able to solve the issues about the standard handover, in fact triggering ourselves the handover we solve the MQTT disconnection phase issue.

At the same time our handover handler solves the RPL reconnection problem because before the new DIO messages from a different network were ignored, as we saw in chapter 4. Now they are accepted as soon as they arrive and they are even filtered to avoid the reconnection to the old network.

This image helps us to understand the steps forward we achieved with this proposal:

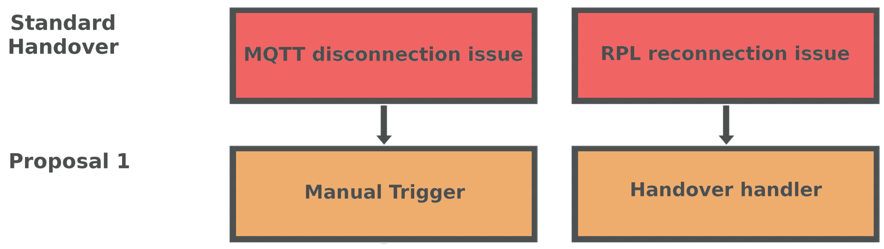


Figure 21: Proposal 1 achievements

But this first implementation, even if we said it is a raw one, lets us think a different way to improve it and make it smarter.

For example from this starting point it would not be hard to think of a handover not triggered by the button pressure, but indirectly by the mobility of the mote, through the RSSI value.

The Re-mote has a sensor to evaluate the Received Signal Strength Indication, as we read in the json messages the node send to the MQTT broker. It is easy to imagine how to build a cycle that every a certain interval of time checks the RSSI value and, if it is under a certain threshold, it starts the handover calling the *my_handover_handler()* function.

Even if this would be an interesting way to proceed, we decided to take another path, that we will show in the proposal 2.

This is because we do not know which variables will be involved in the future in the decision of when performing a handover. This is a huge and different issue and as we said before it is not the first purpose of this project that is focused on how perform it. But it is important to note how it could be easy in a possible future work to take over from here and go down on that road.

5.2.2 *Application layer handover*

As we said in the previous chapter with the second proposal we take a different path.

Now that we have a manual way to trigger the button, and we saw which options could be taken from there, we want instead get to another point: since when trigger a handover is a very big issue, we imagine that this is not a node driven decision, but an application layer one instead.

This could seem some minor difference, but it is a big change in the point of view of the problem indeed. With this change of paradigm we have big advantages.

We do not care any more about the low resources constraints of the devices, because we are taking away from them the handover operation and we are giving it to the application layer.

This means both that our devices have one complex operation less than before to perform and, at the same time, giving the handover activity to the application layer, we are free to think at any possible handover management we want, no matter how complex it could be.

With this different point of view we definitely separate the handover from the device. The handover is now managed by the application layer that can be thought as a complex cloud infrastructure that could be in possession of the whole network information. In this way it can define a proper way to decide *when* make a node perform a handover from one network to another.

In order to do that we need an infrastructure that lets us communicate with the node and ask it to start the operation. We already have it, this structure is MQTT itself.

In fact the broker architecture and the topic messaging model are a suitable option for this purpose.

The changes are in the *mqtt-demo.c* file, in the *pub_handler()* function, that is the function that process the MQTT messages published by the broker in the topic the node is subscribed to.

```
if(strncmp(&topic[10], "hand", 4) == 0) {
    if(chunk[0] == '1') {
      my_handover_handler();
    } else{
      printf("chunk not correct to start the handover.
          Please send a 1 to start the handover \n");
    }
    return;
}
```

Considering that the default topic that the mote subscribes to is *iot-2/cmd/+/fmt/json*, in the example declined in *iot-2/cmd/+/leds/json*, we decided to maintain the same structure to let the new code be easy integrable in the function.

So the new topic to public to is *iot-2/cmd/hand/fmt/json* and the payload that starts the call of the handover function is *1*, defined arbitrary. These messages are easily sent by the MQTT.fx application.

At this moment we only need a value to let the process start, in the future this could be easily an IPv6 address that we can use as a parameter to pass to the rpl-icmp6 file in which we can wait for a DIO from this address.

With this proposal we make another step forward, summarized in the following image:

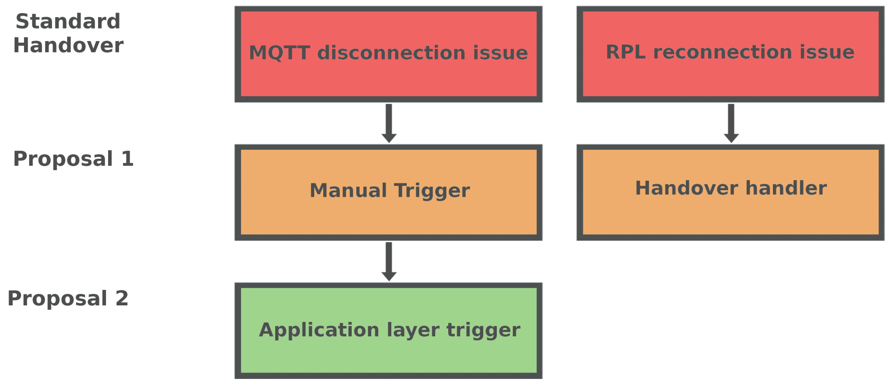


Figure 22: Proposal 2 achievements

5.3 TEST AND RESULTS

So far we discussed the logic improvements of the developed proposal, but to fully evaluate these proposals we have to test their performances to see how they enhanced the standard implementation on the field.

In this chapter we describe how we perform the test and we will discuss the results.

It is important to note that there are no differences between the 2 proposals performances, since they both use the same handover handler. There is a delay from sending and processing the MQTT message, but these activities are out of the handover operation.

Since we are not any triggering the handover by movement, but using both the methods during the test, both manually with the user button and by MQTT message, the test are made under the full cover range of both the border routers. We remember that with the use of antennas the network cover range is about 30 meters, so this is not influencing the tests at all.

5.3.1 *Test implementation*

The purpose of the test is to evaluate the time needed by the system to perform the handover operation, but in order to give a full understanding we do not measure only the total time of the handover, but for all steps of the action.

Therefore the first step of the test is to define checkpoints that we want to keep trace of. To do that we insert some readable *printf* that will be contained in the log, then we will be able to elaborate them.

Checkpoints definition

The checkpoints we define to observe the system are meant to measure any operation involved in the handover and are the following:

1. Button/Message trigger
2. MQTT disconnection
   a) Start disconnection
   b) End of the disconnection
3. Rpl disconnection

   a) Start disconnection
   b) End of disconnection

4. New DIO received
5. New RPL connection
6. New MQTT connection

With these checkpoints we are able to fully monitor the system performances, phase by phase:

- MQTT disconnecting time: defined as the difference between point 2.a and 2.b
- RPL disconnecting time: defined as the difference between point 3.a and 3.b
- Time to receive a new DIO: that is the time from the completed RPL disconnection (3.b), when we send a DIS message, to point 4
- New RPL connection: the time needed to establish a new RPL connection (5) since we receive the new DIO (4)
- New MQTT connection: that is the time from the establishment of the new RPL connection (5) to the new MQTT connection (6), the end of the handover

And from to these measurements we can also have structured measures, probably the most important to us:

- **Total Handover time**: defined as the time from the activation trigger (1) to the new MQTT connection (6)
- **Total RPL Handover**: defined as the time to change the RPL network, since the start of the disconnection from the old border router (3.a) to the connection to the new one (5)

These checkpoints are printed in the node shell, we save the log file through adding the *-l* option when we call the perl script with the command:

```
perl ~/contiki/tools/serial-log.pl -t /dev/ttyUSB2 -b
    115200 -r none -l
```

that saves a file with the default name of serial.log in the directory in which it is executed.

But looking at the log there is still something we need to improve, because the script prints the uptime of the node, but it has a maximum precision in the order of seconds.

This is of course not acceptable for the measurements we want to do and the solution we found was to add a timestamp in the checkpoints using the Contiki function *clock_time()*.

This function measures the uptime of the node in ticks per second, the implementation is platform specific and in the Remote, that uses the Texas Instruments cc2538, we have that *1 tick = 7.8 millisec* according to the datasheet [20].

This is an acceptable precision for our test, to transform the number of ticks we only need to use this easy equation:

```
Number of ticks * 7.8 = Number of milliseconds (/1000 =
    Number of seconds)
```

This is how a checkpoint looks like in the code:

```
printf("CHECKPOINT1 - Handover triggered: CLOCK TIME
    %lu\n", clockmaker());
```

5.3.2 *Results*

The results we present are collected performing 100 handovers with our system.

During the test the handovers are triggered using both the user button and MQTT messages, as we specified, this does not influence in any way the results since both the methods use the same handover handler and we consider as starting point when the handover is started.

It is important to note that the time that elapses between 2 handovers is not significant, since what we are evaluating is only the handover operation. As a method specification, a new handover has been triggered when the previous one ends and the node delivers a message to the broker.

Awk parser

Once the tests are done we collect the log files and we elaborate them with a parser developed in awk [21], an interpreted language designed to work with textual data.

The parser elaborates the log files, it works in different phases:

1. Clean the log: getting only the lines we need, in which we inserted the checkpoints
2. Calculations: using the checkpoints the parser evaluate any handover phase time
3. Average time and standard deviation: after evaluating every checkpoint, the parser return the average time and the standard deviation for any measure

Results

These are the results that the parser elaborates for us:

Total Handover	1293.79	718.28
RPL Handover	442.07	157.898
MQTT Disconnecting	64.71	18.293
RPL Disconnecting	6.11	0.371
DIS to DIO	430.87	157.887
DIO to RPL reconnect	5.09	0.471

Table 5: Results in ticks

The units of measurement of these data are expressed in ticks and the value next to each line is the standard deviation.

To be logically interpreted we need to transform them in seconds and evaluate the other measurements we are interested in:

Total Handover	10,091	5,602
RPL Handover	3,448	1,235
MQTT Disconnecting	0,505	0,143
RPL Disconnecting	0,048	0,003
DIS to DIO	3,361	1,231
DIO to RPL reconnect	0,039	0,003
New MQTT connection	6,138	5,738

Table 6: Results in seconds

Note:
The new MQTT connection is not evaluate directly by the parser, we calculate it as:

```
Total Handover - MQTT disconnecting - RPL Handover
```

5.3.3 *Analysis*

In chapter 4 we saw that the standard implementation of the handover was not satisfying for our purposes because we had an excessive time to perform a handover.

The first result we can comment is how we improved this time with our proposals: we passed from an average time of **163,25 seconds** to **10,09 seconds**.

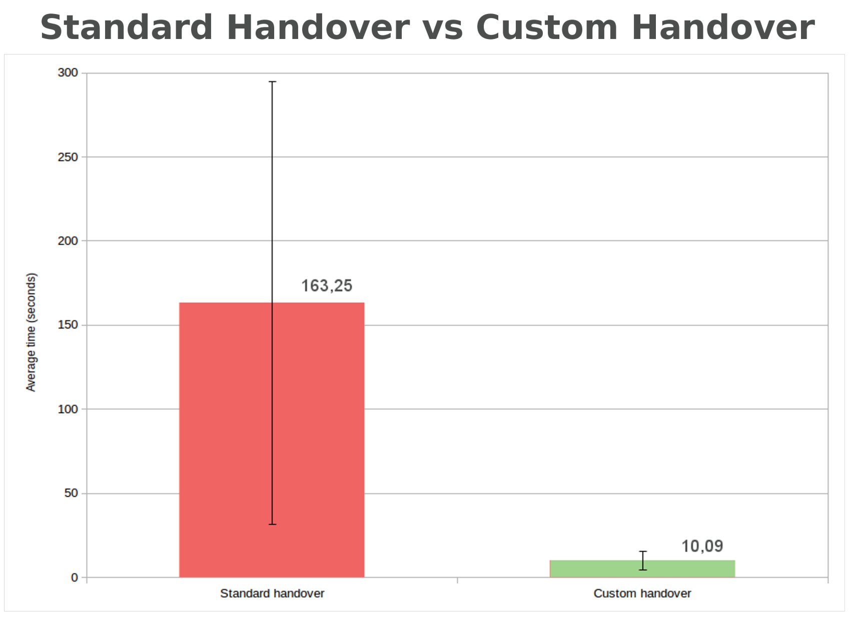


Figure 23: Standard Handover vs Custom Handover

This is a big step forward of more than one order of magnitude (almost two) that proves the work we have done improved sensibly the system performances.

The difference between the two implementations is big as we see, but we want to deeply investigate the result, because 10 seconds in a real scenario is still a big time. In fact these solutions represent a good start, but there is still work that is possible to do on it.

To understand where the further works could be focused it is useful to deeply analyse the results. If we break up the total time measure we can see how this is composed and this offers interesting clues:

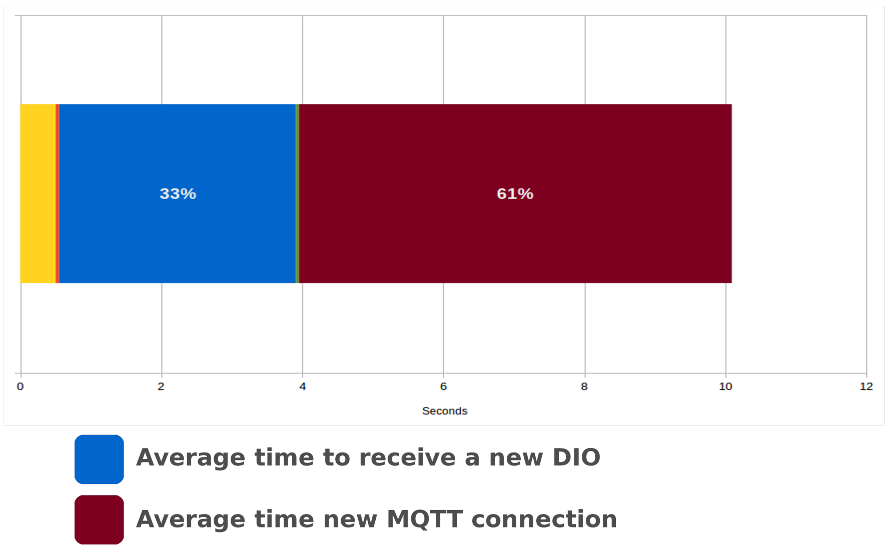


Figure 24: Handover phases comparison

As it possible to see in the image, from the 5 phases that compose the handover operation, 2 of them together represent the 94% of the total time:

- Average time to receive a new DIO: 33% (3,36s)
- Average time new to establish a new MQTT connection: 61% (6,138s)

These results are pretty impressive, it means that the other 3 phases together are only the 6% of the handover, but there is more.

In fact the Average time to receive a new DIO and the Average time to establish a new MQTT connection are even the phases with the larger standard deviation:

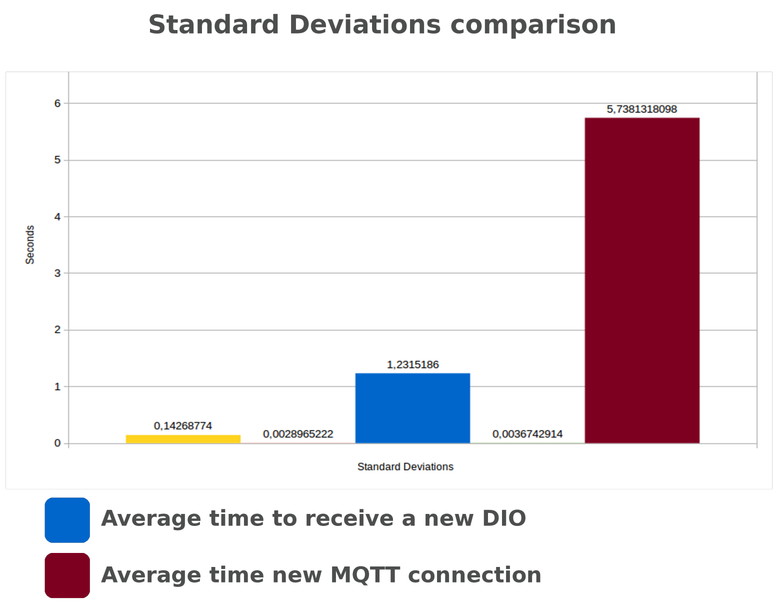


Figure 25: Standard deviation comparison

As a matter of fact, these two phases are the ones in which we have the most of the packets exchange between the node and the border router and on the opposite the phases in which we do not have much packets exchange are pretty faster, like the RPL disconnecting and reconnecting.

In any case, these 2 facts give a strong indication that there is still work to do in those areas.

Work that it is not possible to do inside this thesis, but we will try to give our contribute in the next chapter: Conclusions and Future Work.

6 CONCLUSIONS AND FUTURE WORK

In this final chapter we draw conclusions from the work done, summarizing the most important milestones and we define some clues for the possible future works.

6.1 CONCLUSIONS

The work done in this project can be retraced looking at the main chapters of the thesis: 3 - *Implementation of a Network*, 4 - *Standard Handover* and 5- *Proposals*.

This helps us to individuate the milestones we achieved:

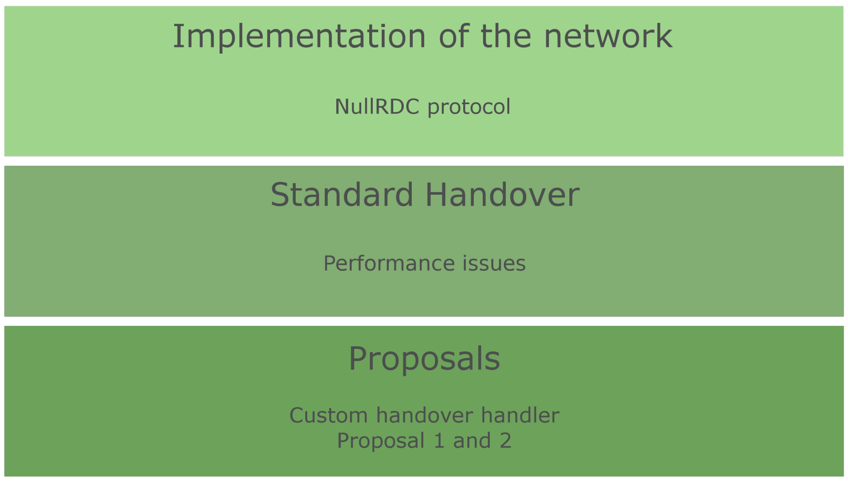


Figure 26: Milestones

Each milestone brings important conclusions on the platform and the work done.

In Chapter 3 we set up the network we used for the rest of the project and in this first step we had the chance to understand the platform we worked on. Even if this was an early stage we had to take important choices as the switch from ContikiMAC to NullRDC.

This change let us have a stable network improving the critical connection phase, with this action we have been able to establish a connection within few seconds compared to the random behaviour we had before and the minutes needed to complete it.

In this way we have been able to go through the project until the end, but the choice we made, even if necessary, has some side effects that have to be considered. The NullRDC protocol has an obvious impact in the energy efficiency of the system and in low resources devices this is a key aspect.

In a possible future real implementation, this aspect, together with the nullmac protocol that comes along, can not be ignored and has to be worked out.

After we had an usable network we studied in Chapter 4 how our system performs the handover operation and we had to conclude that the standard implementation does not provide an acceptable handler.

In fact the bad performance triggers, like timers, events or routines, shows that the handover operation is not well performed in the Contiki platform. This is pretty obvious in the MQTT level, in which we had the node stuck in the publishing phase.

These aspects made clear that there were work to be done and we gave our contribute in the Chapter 5 with 2 different proposals. Each one developed following a different logic.

proposal 1, in which the handover is triggered by the button pressure, is meant to be a base for possible future work that wants to let the node decide on its own when to perform the handover.

On a different point of view, **proposal 2** provide a structure that lets the application layer be in control of the entire system and in charge of the handover operations.

Both proposals are based on the same handover handler that therefore is the core of work presented in this thesis. To develop the new handler we had to go through the whole platform, investigating its deep functioning and trying to affect it as less as possible to preserve its stability, but meanwhile provide a way with acceptable performances to get the work done.

In fact the custom **handler** we developed represent a step forward for the platform, that before was not provided with it and so the handler improves the performances in a sensible way. We again went from a random behaviour to a stable one increasing the measurements by more than one order of magnitude.

In conclusion we can say that the work done can be considered an interesting direction to take for the platform that has now an handler to perform in a stable, performing and logic way the handover operation.

By analysing the tests results it is possible to see that there is still room for improvements both working on the handler itself and both pursuing the 2 different roads defined by the proposals. We provide some clues about that in the next chapter, Future Works.

MQTT deserves a deepening itself, since in the *Introduction* the Research Question number 2 was: *Is MQTT adapt to be on the top layer of our network?*.

At the end of the work we can say that MQTT has more than one issues to work on, for example the longest phase in the handover operation is the MQTT reconnection with the 61% of the whole time. Considering even the absence of an efficient disconnecting routine, these aspects are relevant.

Despite that MQTT showed good points as the unexceptionable messages managing we took great advance of in the proposal 2.

So we can conclude that even though there is work to be done to improve MQTT performances, it can be considered a suitable solution that is possible to use for this project purposes.

6.2 FUTURE WORK

There are different directions in which the future works can be directed and it depends on the purposes that will be achieved.

6.2.1 *Improved Custom handover handler*

Maybe the most interesting future work could be done in the custom handover handler. Even if this is one of the biggest improvements provided by this thesis we found during the work different clues and ideas to work with.

Looking at the custom handover results is possible to notice that one of the two longest phases is the time needed to receive a new DIO from the new border router. If we focus only in the RPL handover, not considering MQTT at the moment, we can see that without this phase is possible to notice how faster could be the RPL handover:

Average RPL Handover	3,448	1,235
Average RPL Disconnecting	0,048	0,003
Average DIS to DIO	**3,361**	**1,231**
Average DIO to RPL reconnect	0,039	0,003

Table 7: Focus on RPL handover phases in seconds

These are the 3 RPL handover phases, without MQTT. We can see that the DIS to DIO phase is the longest one that most affects the total operation. If we could imagine to eliminate this time we could have a faster performance:

```
RPL Handover (3,448146) - DIS to DIO (3,360786) = 0,08736s
```

Less than one tenth of a second is an impressive time, but h**how could be this goal achieved?**

The idea we figured out during the work is about the new DIO messages processing.

At this moment the new DIO messages are ignored by the node while it is connected to another network, they are only accepted once we are not connected. But since we receive them even while we are connected, the possible improvement is to process the new DIO messages even while we are connected to a network and save them in custom variables.

In this way we could imagine that if a smart application layer sends us information about which connection we want to connect to through a MQTT message for example, and we have already a DIO message saved from that network, we can totally avoid the DIO receiving time and directly ask to connect to the new network.

To have a previous DIO saved means that we have been under the cover range of the new network, but since what we want to do is to perform a handover toward this network we should already be, so this is not a problem.

So, as we showed before, we could achieve an RPL handover time under a tenth of a second, even if we have to consider some extra time for the new DIO processing, but this kind of operation is not resources or time expensive.

The possible total time for an handover,considering again the MQTT phases, could become:

```
Total handover time (10,091562) - DIO receiving time
   (3,360786) = 6,730776s
```

An improvement of performance equal to 33,3% and this matches Figure 24.

The logic behind this idea has been fully analysed and looking at the code there are not obstacles known at the moment for its implementation, the most tricky part will probably be managing the DIO variables that need proper structures to be saved and used.

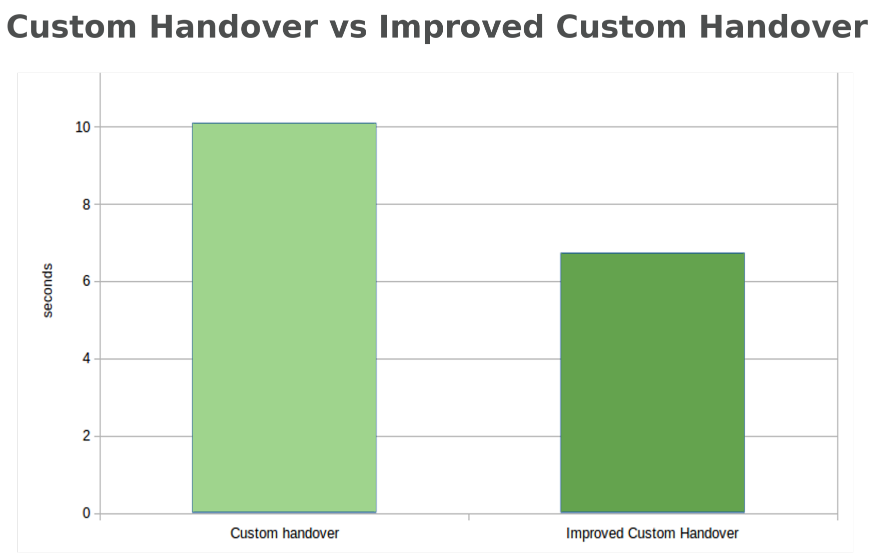


Figure 27: Custom handovers comparison

6.2.2 *Smart application layer*

As we said, in order to pursue this idea we need a smart application layer and this is the main possible future work about the proposal 2.

In fact the second proposal gives us the chance to control the node from a remote position, but at the moment there is no intelligence behind this control, we only trigger the handover sending a MQTT message with the MQTT.fx application.

An interesting work could be done in this area to build a smart application layer, in which we do not have any constraint about the resources or the complexity of the elaboration, that is able to tell the node exactly when perform the handover.
But another important step could be done: the application layer could be in charge not only of triggering the handover but it could even tell the node toward which network performs the handover.

Technically speaking the platform is ready for this step, because at the moment the MQTT messages received by the node that trigger the operation contain a default payload equal to *1*, in the future is not hard to adapt it to receive the *DODAG ID* of the new network for example.

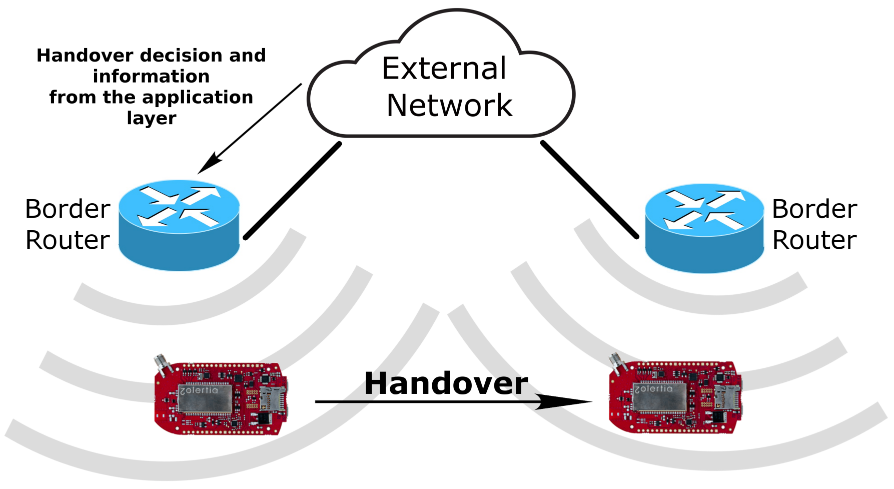

Handover decision and information from the application layer
External Network
Border Router
Border Router
Handover

Figure 28: Application layer handover

6.2.3 *Smart node-driven handover*

A different direction is to improve the node-driven handover, where with node-driven we mean that we do not want to involve the upper layers in the handover operation, but we want to leave the node in charge.

At the moment in the proposal 1 the handover is triggered by the user button pressure, this first step is not directly meant for a real use, even if it is possible, but mostly for giving us the chance to be in control of the handover operation.

It is not hard to imagine how we could improve it checking in a routine some easy reachable environment variables such as the RSSI, the battery level or the RPL routes. By evaluating them the node could decide to trigger the handover itself.

Since the node is meant to have low resources, this future work should be directed in the evaluation of simple environment variables that for example can be compared to defined thresholds.

Even if this approach is simpler than the previous one, it can have better performances since everything is done in the node, and it could be used in some emergency case in which we want the node to be suddenly able to react to some environment change as going out of the cover range or a broken border router.

6.2.4 *Mixed approach*

As is possible to see, both the directions, the application layer or node driven-handover, have good points and are suitable for different purposes.

This suggests us what could be the final step: a mixed approach.

What we mean for mixed approach is to have a smart application layer in charge of most of the activities, that has a total knowledge of the networks and with a smart algorithm to decide when and toward which network the handovers should be performed.

At the same time we could have on the node some basic routine to handle the most critical situations such as a missing border router. As previously explained, these checks can be done with thresholds that are not resource demanding.

In this way we could have a very efficient system that handles most of the activities from the application layer, in which we have not resources constraints, and the emergencies handled directly in the node to be always ready to answer to the environment changes.

The following image show this idea:

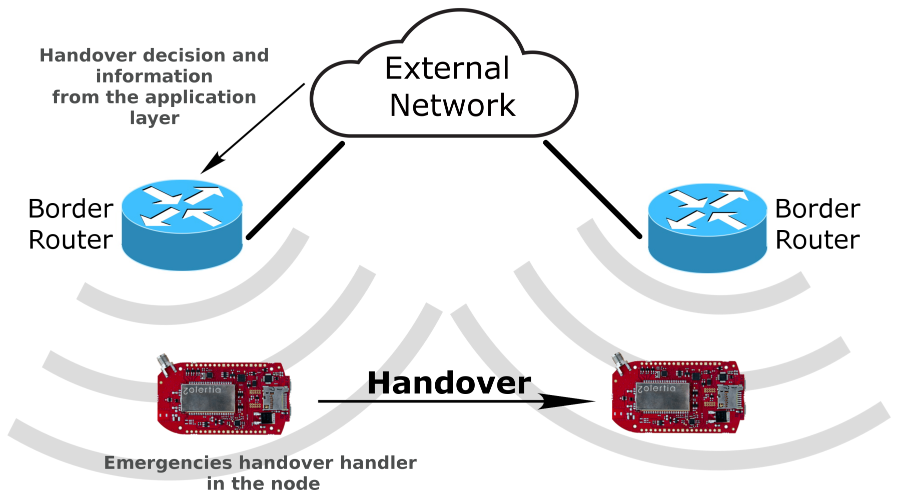

Handover decision and information from the application layer
External Network
Border Router
Border Router
Handover
Emergencies handover handler in the node

Figure 29: Mixed approach

In conclusion we can say that all these possible future works are proofs that the work presented in this thesis constitute a new, stable and flexible base that can represent the first step for future improvements.

BIBLIOGRAPHY

[1] IEEE 802.15.4, 2011.
https://standards.ieee.org/getieee802/download/802.15.4-2011.pdf

[2] Wi-Fi Alliance website.
http://www.wi-fi.org/

[3] Bluetooth technology website. (Last access: 12/04/2016)
www.bluetooth.com

[4] Internet Protocol, Version 6 (IPv6) Specification, Network Working Group.
https://www.ietf.org/rfc/rfc2460.txt

[5] IoT in 5 days, Github repository. (Last access: 28/05/2016)
https://github.com/alignan/IPv6-WSN-book

[6] IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. Network Working Group
https://tools.ietf.org/html/rfc4919

[7] RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. Internet Engineering Task Force (IETF)
https://tools.ietf.org/html/rfc6550

[8] Contiki website. (Last access: 22/05/2016)
http://www.contiki-os.org/

[9] Thread Group website. (Last access: 18/04/2016)
http://threadgroup.org/

[10] MQTT website. (Last access: 12/06/2016)
http://mqtt.org/

[11] Mosquitto website. (Last access: 20/05/2016)
https://mosquitto.org/

[12] Zolertia Github repository. (Last access: 24/06/2016)
https://github.com/Zolertia

[13] Zolertia website. (Last access: 15/04/2016)
`http://zolertia.io/`

[14] Contiki Github wiki, "Processes" section. (Last access: 21/05/2016)
`https://github.com/contiki-os/contiki/wiki/Processes`

[15] Sensniff github page. (Last access: 11/05/2016)
`https://github.com/g-oikonomou/sensniff`

[16] Whireshark website. (Last access: 16/06/2016)
`https://www.wireshark.org/`

[17] MQTT.fx website. (Last access: 25/04/2016)
`http://mqttfx.jfx4ee.org/`

[18] Using RPL. (Last access: 29/06/2016)
`https://github.com/maniacbug/contiki-avr-zigduino/wiki/Using:RPL`

[19] Contiki Github, Radio Duty Cycle Protocols. (Last access: 07/07/2016)
`https://github.com/contiki-os/contiki/wiki/Change-mac-or-radio-duty-cycling-protocols`

[20] CC2538 Microcontroller System-On-Chip, Texas Instruments. (Last access: 05/07/2016)
`http://www.ti.com/lit/ds/symlink/cc2538.pdf`

[21] The GNU Awk Users Guide. (Last access: 14/07/2016)
`https://www.gnu.org/software/gawk/manual/gawk.html`