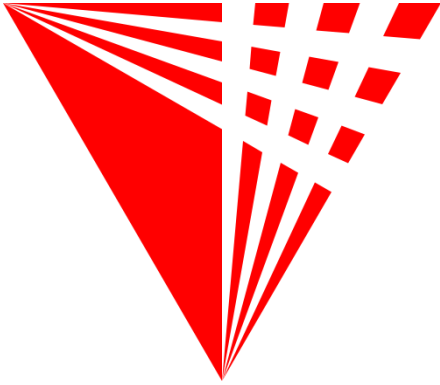


Illinois Institute of Technology



ECASP

Master of Science in Electrical Engineering

Year 2016/2017

**Implementation of 6LoWPAN (IPv6) for
Wireless Body Sensor Networks (WBSN)**

Fernando Javier Balseiro Lago

Supervisor: Dr. Jafar Saniie

Abstract

This research report will describe the work that has been done in implementing a *Wireless Body Sensor Network* (WBSN) using a communications protocol based on 6LoWPAN. Basically, 6LoWPAN is an adaption layer which is able to make IPv6 packets compatible with networks based on the IEEE 802.15.4 standard. This way, these networks can take advantage of the multiple benefits that working with IP technologies brings e.g. interoperability with any other IP or IEEE 802.15.4 devices.

Using this technology, our research will focus on implementing a WBSN that will be capable of monitoring the state and condition of a patient that is outside the premises of a health institution. In order to do so, we will attach sensors to the patient's body and have these send the sensed data to a processing gateway. The gateway will then handle the data in order to extract valuable information and send it to a server database hosted in an IP address. The data will be then accessed from any hospital avoiding the necessity of displacing medical staff to the premises of the patient for checking his/her condition.

Table of Contents

Introduction	1
<i>Context of the Project</i>	<i>1</i>
<i>Description of the Objectives.....</i>	<i>2</i>
<i>Structure of Report.....</i>	<i>3</i>
Background	5
<i>Hardware Equipment.....</i>	<i>5</i>
<i>CC2650 SensorTag</i>	<i>5</i>
<i>Raspberry Pi 2.....</i>	<i>6</i>
<i>Standard/Protocol Description:</i>	<i>7</i>
<i>IEEE 802.15.4</i>	<i>8</i>
<i>6LoWPAN.....</i>	<i>9</i>
<i>CoAP</i>	<i>10</i>
Project Development	11
<i>Set Up the Sensor Node Network</i>	<i>11</i>
<i>Configuring the WBSN.....</i>	<i>12</i>
<i>Radio Interface through SensorTag</i>	<i>12</i>
<i>Radio Interface through OpenLabs Module</i>	<i>14</i>
<i>Operating the WBSN.....</i>	<i>22</i>
<i>Deploy Processing Algorithms in the Raspberry Pi 2</i>	<i>24</i>
<i>Position Tracking of the Patient.....</i>	<i>25</i>
<i>Sitting and Lying Down Detection</i>	<i>30</i>
<i>Running Detection.....</i>	<i>31</i>
<i>Falling and Lying Orientation</i>	<i>33</i>
<i>Configure a User Friendly Application on the Server PC.....</i>	<i>35</i>
Overview & Results	36
<i>Overview of the Health Monitoring Application</i>	<i>36</i>
<i>Results of the Health Monitoring Application</i>	<i>38</i>
Discussion	41
<i>Discussion of Results</i>	<i>41</i>
<i>Discussion of Enhancements</i>	<i>42</i>
Conclusion	46
References.....	47

Introduction

In this first chapter, we will offer an introduction to the research project. We will start off by describing the context of our project and the different reasons on why we think our research will be important in the field of *Wireless Body Sensor Networks* (WBSN). Once given this contextualization, we will describe the aims that we pursue in our research as well as a description about the structure of the project.

Context of the Project

The concept of Internet of Things (IoT) is becoming more and more popular within the networking field of study. Basically, IoT allows the inter-networking of devices such as sensors, actuators, vehicles, etc.... to exchange data amongst them and have access to the network. This last property will allow users to both monitor and control remotely devices through the Internet infrastructure. This will improve both the efficiency and accuracy in the interactions since there is no need for the user to be physically present in the sensor's site to collect data or set instructions.

IoT has been able to expand and consolidate due to several factors. Amongst many, the most notorious one has to do with the fact that broadband Internet access is becoming more available than before [1]. Today we are continuously connected to the web, either by our computers or by our cellphones through the mobile network infrastructure. Due to this continuous connection to the Internet, being able to monitor and control devices remotely at any time is something that a user can gain a benefit from. Also, technology cost reductions have been a significant factor for the growth of IoT networks. More devices are being manufactured than before and, due to technology advances, these are able to convey additional and more advanced functionalities. For instance, physical sensors are now able to recollect data and, at the same time, send it through radio-frequency to another node for later process. These have been factors that have set the right environment for IoT networks to grow and consolidate as an idea in today's society.

In this context is where an idea arises and which will constitute the basis of our project. One of the benefits of IoT networks is their ability to monitor 24/7 the data extracted from networks on an off-campus site [2]. The collected data could then be accessed ubiquitously by medical staff who are working in the hospital and thus perform remote diagnostics of patients who are in their house. The advantages of this set up are many amongst which we can highlight [3] [4]:

- **Remote diagnosis:** the medical staff does not need to move outside the hospital's premises in order to perform surveillance over patients. This saves valuable time for doctors since the time spent in displacing them to the patient's location can now be used to treat other patients.
- **Remote Follow-ups:** with this new system, doctors can perform follow-ups on people who are at their homes. This means that the medical staff could now release patients earlier from the hospital's site leaving room for more patients. This will reduce healthcare costs greatly and also improve the service of hospitals.
- **Automated system:** the system proposed comprises the advantages of a computerized system. Now instead of the medical staff evaluating the patient,

writing their vitals and then introducing them into the computer system we can omit these steps and let the computer handle everything. This system will record the readings performed by the sensors without the intervention of the medical staff, reducing consequently aspects such as human error.

- **Alert Based System:** since there will be a processing node in the system proposed, this one could analyze data and send alarms to the medical staff. This means that now doctors could be warned if something was wrong with a patient and therefore increase the on-time interventions of these.

From the above we can see that the benefits an IoT network brings to the health monitoring of patients are numerous. Therefore, our project will be based in deploying an IoT network whose ultimate goal will be for medical staff to check patient's vitals remotely.

Description of the Objectives

Once given a description of the context and the reasons why we want to develop the project proposed, it is now time to mention the objectives we aim to achieve in our research. Before getting into details, it is important to give a basic explanation of what the architecture of our system will be like. Once described, we can concisely state the objectives we proposed for our work.

In figure 1 a basic architecture is presented. The hardware elements from which they are made of, the Raspberry Pi 2 [5] and SensorTags [6], will be described afterwards leaving for this part the main networking concerns and general operation.

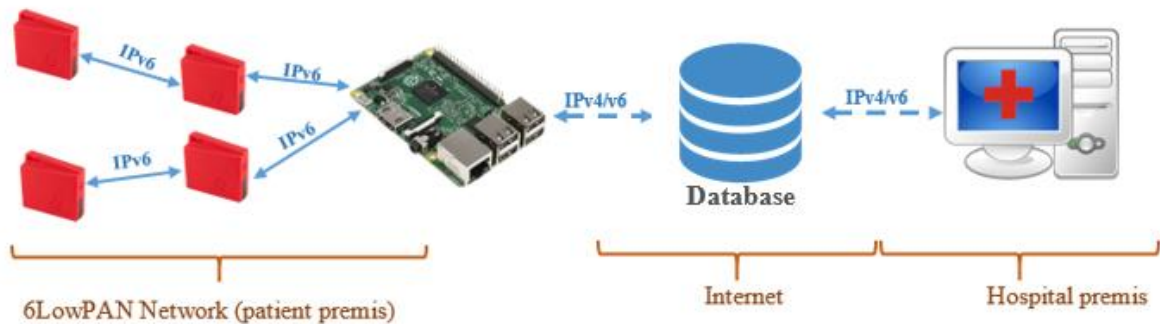


Figure 1: Architecture of the System

As we can see from the picture above, the SensorTags within the patient's premises, will communicate with a Raspberry Pi 2. This platform will act as a gateway which will process the raw data gathered from the SensorTags and send a message, with all the relevant patient's information, to a server database which will be online 24/7 and hosted on an IP address. The server database will have its data fetched by a client application deployed on a PC within the hospital's premises. This way, the client application won't need to be always online or store large quantities of data.

It needs to be noted that a scenario where the sensors exchanged data with each other is contemplated in the systems' architecture. This is because IEEE 802.15.4, which defines the physical and media access control layers for personal area networks (PANs), has a very limited range. In order to counter this problem, the network protocol that sits on top along

with its adaption layer for IEEE 802.15.4 usage, IPv6 with 6LoWPAN, allows multiple hops for the data to reach the gateway.

This will make up the general architecture we wish to implement in our ubiquitous health monitoring application. In order to deploy this system successfully, we will approach this project in a set of objectives:

- **Set up the sensor node network:** research alternatives of sensors capable of measuring relevant magnitudes for health monitoring (such as temperature, accelerometer information...). Once found the devices, we will program them conveniently in order to deploy a network.
- **Configure the WBSN:** the aim is to set up the 6LoWPAN network that will be residing in the patient's premises. We will try and search for alternatives that will allow the Raspberry Pi 2 to act as a gateway between the two networks: the Internet and the WBSN. The alternatives contemplated, as it will be seen later in the report, are strongly determined by the hardware device used for the transceiving device between the Raspberry Pi 2 and the SensorTags.
- **Implementing the WBSN:** once we are sure that there is communication between the Raspberry Pi 2 and the SensorTags, we can implement the WBSN. This objective will mainly try and see if the Raspberry Pi 2 is able to extract data (temperature, accelerometer data...) from the several SensorTags adhered to the patient's body. It has to be noted that communicating with more than one SensorTag will imply generating some type of scheduling amongst the sensors for an organized transmission of data.
- **Deploy processing algorithms on the Raspberry Pi 2:** we will implement processing algorithms in the Raspberry Pi 2 in order to exploit the advantages of having a gateway rather than simply sending the raw information to the server database. The aim of this objective is that, by handling the magnitudes provided by the SensorTags, we will be able to extract valuable information regarding the condition and state of the patient.
- **Configure a user friendly application in a server PC:** the last objective is to be able to communicate the Raspberry Pi 2 to an application deployed in a PC to which the former one will send the processed data. This application will be the server database previously mentioned where all the pertinent information regarding the patient will be stored.

The above objectives have been set as steps that will ultimately lead to the main purpose of our research: implementing a ubiquitous health monitoring application that uses 6LoWPAN for communication purposes amongst the sensors.

Structure of Report

This report will firstly provide a *Background* section to the reader. The objective of this part is to give the reader a detailed description about all the hardware and software tools that will be needed in order to carry out the research.

In the third chapter, we will give a description of all the different procedures and challenges that we have encountered when working in our research. The structure of this part will be easy to follow since each of the subsections will be each of the objectives declared. For the first set of objectives we will provide, along with the procedures, results to show that in fact the objective has been met. On the other hand, for the last two objectives the results will be provided in the next section as they are believed to be the main ones of this research.

The *Overview & Results* part will start by providing the reader a general overview of how our application will flow throughout its execution. The aim of this is to convey in one flow graph all the functions our application will be able to perform. Once the reader knows the different functionalities, we will give proof to demonstrate the correct operation of our health monitoring application.

From the results presented in the section above, in *Discussion* we will do an objective evaluation on them. We will firstly give arguments on why our application is not perfectly functional and give different procedures on ways to improve it. We will also discuss other functions that we could implement in our system in order to enhance the operations of our health monitoring application.

Lastly, the *Conclusion* will close the report by summarizing everything that has been described.

Background

In this part, we will talk about the different software packages and hardware equipment needed in order to conduct the project, as well as give a description on several important standards and protocols that will be utilized. We will start off by describing the hardware that has been used in this research.

Hardware Equipment

Regarding the hardware used, there are two main elements: the CC2650 SensorTag and the Raspberry Pi 2.

CC2650 SensorTag

The SensorTag platform uses a CC2650 SimpleLink™ Multistandard Wireless MCU [7] whose block diagram can be seen in figure 2. This device is targeted for wireless communications such as Bluetooth, Zigbee and 6LoWPAN. Below we give a brief overview of its most important features:

- MCU based on a 32 bit ARM Cortex M3 processor that uses a clock speed of 48-MHz.
- Memory Structure: 128 KB of in-system programmable flash, 8 KB of SRAM for Cache purposes and 20 KB of ultralow-leakage SRAM.
- The IEEE 802.15.4 MAC is embedded into the ROM and for its implementation a separate processor is used, an ARM Cortex-M0 processor. This independency allows an improved system performance.
- Some of the most important supported peripherals and modules that will be important for our research will be the UART, I²C, an RTC and a temperature/battery monitor.

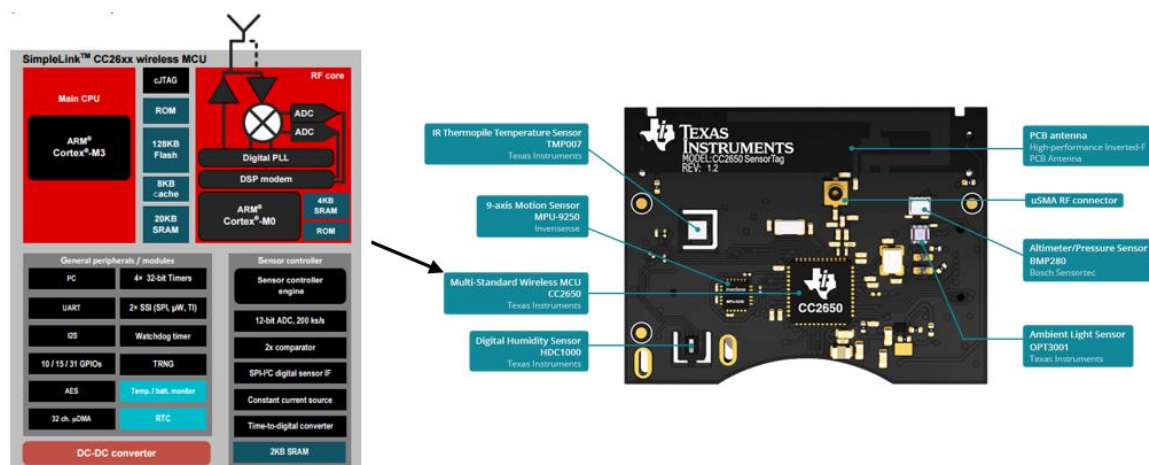


Figure 2: (left) Block Diagram of CC26xx MCT (right) Functional Diagram of SensorTag

This MCU will be mounted on a platform that will enhance the magnitudes read by the sensors. As seen in the figure above, there have been implemented several sensors in the

board and in the table below we give a brief summary of them. Now in order to retrieve the information available from these sensors, the communication between them and the processing unit will be done through the I²C bus. It has to be said that the MPU9250 will use a separate communications bus from the rest of the sensors as well as a separate source of power.

Sensor	Name	Data Size	Manufacturer
IR temperature	TMP007	2x16 bits	Texas Instruments
Humidity	HDC1000	2x16 bits	Texas Instruments
Movement	MPU9250	9x16 bits	Texas Instruments
Optical	OPT3001	1x16 bits	Texas Instruments
Pressure	BMP280	2x24 bits	Bosch

Table 1: Sensors Present in the SensorTag

It is true that in both the processor and the board there are two temperature sensors, however they each measure different parameters. While the IR temperature present in the board measures ambient temperature, the one existing in the MCU measures the temperature of the battery; useful for avoiding overheating. The data read from these sensors will then be transmitted using IPv6 over the IEEE 802.15.4 MAC layer through the use of 6LoWPAN.

Raspberry Pi 2

From figure 1, we can see that a gateway is needed if we want the data of the sensors to be accessed through the Internet. In order to implement this gateway, we will use a Raspberry Pi 2. This device has been used for several reasons amongst which we can highlight its low cost along with its high computational power. Also, the great existing documentation and the projects developed in this platform will help us when troubleshooting our system.

The Raspberry Pi 2 does not alone provide an IEEE 802.15.4 radio interface, therefore additional equipment was needed. In order to provide this connectivity interface, two possible alternatives were studied which their greatest difference resides in the piece of hardware used.

The first one, which was initially implemented, is a SensorTag which is flashed with a specific program and connected to one of the USB sockets of the Raspberry Pi 2. Thus the SensorTag will be the device that provides the RF interface for the 6LoWPAN wireless communication between the sensors and the Raspberry Pi 2. It has to be said that, for this approach to work, we needed to install in the Raspberry Pi 2 a software with the name of 6lbr. This tool, which will be discussed later, basically configures the Raspberry Pi 2 to act as a 6LoWPAN/RPL border router. This way, devices outside the WBSN but with Internet connectivity are able to communicate with the sensors at a network level.

One of the advantages of this approach is that it is easy to deploy and therefore it will be a good starting point for building our system. Another of the advantages is that it is a well-known solution and therefore there exists numerous sources of documentation to consult if problems arise. On the figure below on the left, we show how the connection between the Raspberry Pi 2 and the SensorTag will be done. Notice how a component known as the DevPack Debugger is connected on top of the SensorTag to interface with the Raspberry Pi 2 through the USB interface.



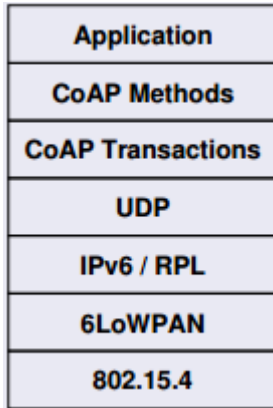
Figure 3: Radio Transceiver Alternatives using a (left) SensorTag or (right) OpenLabs Module

An alternative to the previous method exposed is connecting a module known as OpenLabs [8] which allows the usage of 6LoWPAN, shown on the figure above on the right. This module is connected to the Raspberry Pi 2 through the GPIO pins. The problem with this module, as it will be discussed in the *Project Development* section of this report, is the installation process. In order for this module to be detected by the Raspbian Lite distribution running on our Raspberry Pi 2, we will need to modify accordingly its Linux kernel and recompile it. Also it has to be said that, since the 6lbr application needs to have the radio interface connected through the USB and not the GPIO pins, configuring the Raspberry Pi 2 as a border router is a harder task following this approach.

However, despite the disadvantages mentioned, we have chosen to use the OpenLabs module as a radio interface. There are several reasons for this, starting with its efficiency. The OpenLabs module is a device that has been fabricated for the only purpose of operating as an antenna while the SensorTag has not. Therefore in transceiving operations, it is really likely that the OpenLabs module will outperform the SensorTag. The second reason is the difference in cost in where the SensorTag's price is 30 \$ while the OpenLabs module only 10 \$. Lastly, the documentation existing on deploying a 6LoWPAN network with the OpenLabs module and the SensorTags as sensing devices is practically inexistent. Therefore, by achieving this goal, we will have a project of greater value for the research community.

Standard/Protocol Description:

The sub sections above provided an overview of the hardware elements that will be needed in order to build the system in mind. Now it is time to discuss briefly the standards and protocols that will allow the wireless communications.

IEEE 802.15.4*Figure 4: Network Stack*

The IEEE 802.15.4 is a standard that defines both the physical and MAC layers whose purpose is to be able to support low cost, speed and power communications between devices. It can be comparable to the Wi-Fi protocol only that this one uses more bandwidth and thus requires more power. Below we will give a general overview of this standard showing in figure 4 on the left the network stack that will be implemented in our system.

IEEE 802.15.4 [9] is greatly utilized for the communications of embedded devices. One of the ultimate applications of this standard is to be able to support the networking of a series of sensors attached to a person. This type of scenario involves certain aspects that will have to be implemented in the physical layer. For instance, it is important that these sensors, during the communications with each other, employ as low power as possible; they will run on a battery

which has a limited power capacity. The physical layer implemented by the discussed standard guarantees low power usages however at the expense of reducing the transfer rates, 20-40 kbps, and communications range, 10 meters.

The physical layer of the IEEE 802.15.4 is responsible for managing the radiofrequency transceiving operations, including selecting the channel upon which the transmission will take place. The frequency bands are defined differently in USA and Europe; for the first one there are ten channels in the 902-928 MHz range while in the second one there is one channel in the range of 868-868.6 MHz. It needs to be said that in these channels, the data rate used in transmission will be in the order of 20-40 kbps. However, IEEE 802.15.4 also defines another range of frequencies, 2400-2483.5 MHz, in which the data rate of transmission can reach values of 250 kbps having as a downside the use of more power.

On the other hand, we have the MAC layer defined in the standard whose main purpose is to transmit the frames through the physical layer previously defined. Amongst the most relevant functions that it implements we have the collision avoidance through CSMA/CA. The operation of this algorithm will inevitably see its effect in power consumption which will reduce the battery life of the system deployed.

It has to be said that the IEEE 802.15.4 defines a network topology which uses two types of network nodes. The first type, is known with the name of full-function device (FFD) and it will basically act as the coordinator of the personal area network. In order to do so, this node will have to implement a fully functional IEEE 802.15.4 stack. The other type of devices are reduced function devices (RFDs) and they are really simple devices in which a simple IEEE 802.15.4 stack is implemented. These nodes are not able to coordinate the PAN and therefore they only perform operations such as communicating with the FFDs.

These devices mentioned earlier, form networks that allow the exchanging of data between the FFD and the RFDs. There are multiple topologies in which these nodes can be organized into such as the ones shown in the picture below.

*Figure 5: (left) Star (Right) Peer-to-Peer Topology*

The star topology, shown on the left, is an example in which all of the nodes communicate with a central FFD node. The communication overhead in this node is very large when compared to the other nodes of the networks and therefore it is normally connected to a reliable source of power. On the right we have a peer-to-peer communication in which the nodes can communicate directly to the central node or through other point-to-point links. This topology has the advantage that the processing of data can be assigned to other nodes, however this means that more than just one FFD should be in the network.

6LoWPAN

Following the stack shown in figure 4, it is time to talk about the adaption layer known as 6LoWPAN.

There have been many motivations behind trying to use IP based network layers over IEEE 802.15.4 for WBSN [10]. One of these is the extensive existing work done based on IP technologies which will allow the WBSN to use most of the already defined structure. However, the most important characteristic is that devices using IP network layers can be immediately connected to other IP based networks, say for instance the Internet. This implicit compatibility allows the data to travel through the Internet without the need of intermediate structures that perform translations, e.g. proxies. This way we can easily access data ubiquitously.

Despite the inherent advantages mentioned earlier, there is one main problem in this. The IEEE 802.15.4 supports a frame size of 127 octets only. Taking into consideration the header sizes used in IPv6 and UDP, the frame size supported by the physical layer is insufficient [11]. Therefore, there needs to be a compression in the data packets that allow for the transmission of IPv6 packets to be done in fixed size frames of IEEE 802.15.4. In this context is where 6LoWPAN becomes necessary.

6LoWPAN was defined with the purpose of using IPv6 over the power efficient IEEE 802.15.4 physical layer. In order to do so, this adaptation layer conveys many compression and fragmentation/reassembly functions. Below we give a review of the functions performed by 6LoWPAN:

- **Adapting packet sizes:** as mentioned earlier, 6LoWPAN is responsible for performing the correct operations that will allow the packets that come through the WBSN to be routed to an IPv6 network. This involves packet processing functions such as compression of the IPv6 headers and fragmentation of the packets.
- **Address resolution:** there are several ways for nodes to acquire IPv6 addresses. The one that will be used in our system is the derivation of this unique addresses through the PAN-ID of the network and its MAC value.
- **Security:** the 6LoWPAN adaption layer has to take into consideration whether security algorithms, such as AES, are being implemented in the IEEE 802.15.4 layer. If they are, the adaption layer will have to perform the adequate operations for compatibility issues.

At this point, we have discussed the usage of IPv6 for communicating the sensor nodes amongst each other. However IP protocols require some sort of routing algorithms in order to reach certain destinations. Normal routing algorithms used tend to consume a lot of power, they are processes that rely much in iterative operations. Therefore, a power efficient routing

algorithm has been proposed in order to be implemented in WBSN. This routing algorithm is known as the Routing Protocol for Low-Power and Lossy Networks (RPL).

The RPL algorithm uses instances known as Destination Oriented Directed Acyclic Graphs (DODAGs) in order to perform their traffic routing. Traffic in this type of graphs flows either upwards, towards the root, or downwards, towards the leaves of the graph. In the below figure it is illustrated three DODAGs pertaining to an RPL instance.

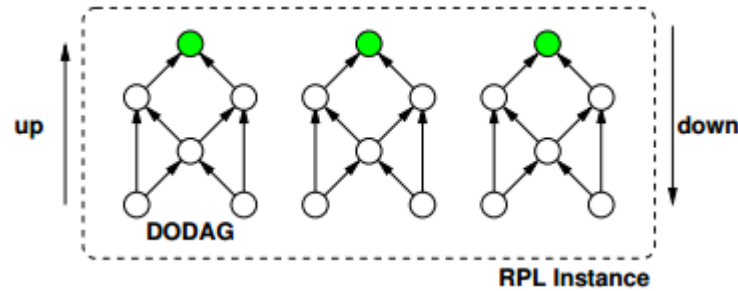


Figure 6: Three DODAGs Pertaining to a RPL Instance

Each node in a DODAG has a rank value which basically states its relative position from the root node, having this one a value of 0. In order to perform the routing, or construct the DODAG instance, the nodes exchange periodically control messages known as DAG Information Objects (DIO). The nodes will then listen for these messages and decide whether to join the DODAG advertised or stay in the one they are at. Based on this information, the node will then be able to choose the parents, nodes with a lower rank, to reduce the cost to reach the DODAG root.

CoAP

Lastly, we will talk about the Constrained Application Protocol (CoAP) [12]. This is an M2M web protocol whose use has been intended for its deployment in inexpensive devices. This protocol is ideal for IoT networks since it is able to function in nodes that have as low as 10 KB of RAM and 100 KB of code space. The CoAP is very similar to an HTTP web transfer protocol since they are based on a REST model. In this type of modelling, servers will provide their resources through a specific URL and then the clients will access them via the known GET, PUT, POST and DELETE methods.

Looking back into figure 4, CoAP transactions allow UDP messaging to be reliable. This CoAP implementation in our system will be of use in the following sense. In each of the sensor nodes, we will implement a CoAP server. This server will make available its resources through an URL. Now, in the Raspberry Pi 2 we will write a CoAP client which will basically make requests to these CoAP servers. The requested data will be then transmitted wirelessly, through the stack of protocols described earlier in this section, to reach the gateway. The gateway will receive the data and process it in order to acquire relevant information from it.

This concludes our background information regarding the project developed for this research.

Project Development

In this part of the report we will describe the different procedures followed in order to meet the objectives we have set in our research. Each methodology will be accompanied by a set of challenges which will be indicated within the section. This part will be divided into five sub-sections, each one pertaining to one of the objectives previously listed. The structure followed has been thought to be ideal since the objectives by themselves constitute a step by step progression into implementing a health monitoring application, which is the main goal we are trying to achieve.

It has to be said that for all of the objectives except the last two, screenshots of the results will be provided. This is because these last two comprehend the actual implementation of the health monitoring application, our final goal. The results gathered from the rest of the objectives are just intermediate steps in order to achieve the final aim of our investigation.

Set Up the Sensor Node Network

This is the first objective of our research and it will mainly focus on choosing the device that will be adhered to the body of the patient and act as a sensor. Now, since this objective is closely linked to an investigation process, searching alternatives through different sources, its extension won't be long. We will start this sub-section by giving the reasons behind using the SensorTag CC2650 as the sensing device.

First of all, the price of this device is fairly cheap, it's only 30 \$. Therefore by utilizing it for our system, we will be generating an application which is affordable for its implementation. Another reason is the great presence of documentation and projects regarding this device and which will surely help us in the future when fixing problems in our system. However, the most important reason out of all resides in how simple it is to program these devices once the environment is properly set in our workstation.

For programming the SensorTags we will need a Virtual Machine with the name of Instant Contiki, necessary in order to work with Contiki. Contiki [13] is an open source operating system which its intention is to run on low-power microcontrollers. Thanks to the use of this operating system, devices can use efficiently their hardware and provide low-power communications. For instance, Contiki uses an open source implementation of the TCP/IP stack known as uIP which is intended for low-demanding devices such as the SensorTags we will be using.

Now by downloading the Contiki Virtual Machine in our PC, we will be provided with all the tools necessary to compile and build programs for this operating system. Also, the Virtual Machine includes a repository of examples from which we will be able to learn both how to code and how to use the functions provided in Contiki in order to develop applications of our own. Out of all the examples available, the one that will have a great importance in our application is the one named *cc26xx-web-demo.c*. The source code contains all the configuration needed to deploy a CoAP server which makes available all the sensor data for incoming requests from a CoAP client, our Raspberry Pi 2.

In order to generate the .elf file to flash the SensorTag with [14], we will need to execute a make command in the directory where the *cc26xx-web-demo.c* source code is. In this case, the command to generate the .elf file is the following.

```
sudo make TARGET=srf06-cc26xx BOARD=SensorTag/cc2650 cc26xx-web-demo.bin  
CPU_FAMILY=cc26xx
```

The above command basically defines for the compiler both the target and the board on which the example will be deployed and its CPU family. This way, the compiler can make the .elf file accordingly to the platform in which it will run.

Once the proper .elf file is generated, we will need a tool that will allow us to actually flash these programs in the CC2650 SensorTag. For this process we will use Uniflash, a software branched from *Code Composer Studio* which will allow us to flash microcontrollers from Texas Instrument vendors. Note that for the flashing process we will also need the DevPack Debugger to connect the SensorTag to the PC.

At this point of the research we both know how to write programs for the SensorTags as well as flash them. Although information regarding on how to write code in Contiki has not been provided [15], since it is believed to be out of the scope of the report, we can mention one important aspect. The use of *#defines* in programs written in Contiki is abundant. These defines are present in both the source code and a header file with the name of *proj_conf.h* and they basically declare relevant information such as the PAN ID of the sensor or even the time period which sensor data is refreshed. It will be important, as it will be seen later, to pay attention to these configuration parameters.

The above is a general description of how to program the SensorTags that will form part of the WBSN. With this part done, it is now time to escalate within the architecture of our system and start configuring what will be our gateway to outside networks.

Configuring the WBSN

This objective constitutes the first step in achieving our ubiquitous health monitoring application. From a networking point of view, it is the most important step since its fulfillment implies providing the Raspberry Pi 2 with a radio interface through which it will be able to communicate with the SensorTags via 6LoWPAN and immediately act as a gateway to outside networks through the Internet. Now in order to acquire this interface, as we mentioned earlier, we have two alternatives.

Radio Interface through SensorTag

For this alternative to work, we first need to install a set of software packages in our Raspberry Pi 2. Amongst many, the most important one is the application, already mentioned, 6lbr [16] which configures the Raspberry Pi 2 as a 6LoWPAN/RPL border router.

Once 6lbr was installed, we had to choose one of the many modes 6lbr is able to run in. The three main categories are bridge, router and transparent bridge each of which can be selected by adding the correct configuration lines in a file called *6lbr.conf* present in the */etc/6lbr* directory. We decided to choose bridge as the configuration of the border router since this mode allows the Raspberry Pi 2 to have connectivity to the 6lbr application. In order to do this, a virtual Ethernet interface with the name of 'tap0' is created. Through this interface, we will be able to access the sensors in the WBSN from either the Raspberry Pi 2 or a PC directly connected to the Ethernet port. Note that the direct Ethernet connection between the configured border router and the computer is used in the meantime for debugging purposes, the ultimate objective will be to connect this Ethernet cable from the Raspberry Pi 2 to a router providing outside networks connectivity to the WBSN.

It has to be said that in order to connect the tap0 interface with the Ethernet port of our Raspberry Pi 2, a br0 interface is used. This interface will bridge both eth0 and tap0 interfaces. In the figure below we can see a diagram which illustrates the above configuration.

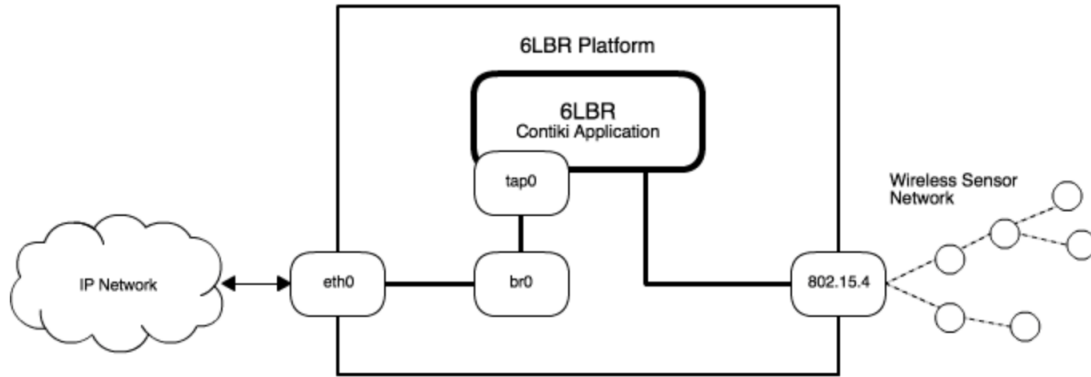


Figure 7: Bridge Configuration of the Border Router

At this point, we have a border router deployed in our Raspberry Pi 2 that will give connectivity to outside networks, such as a PC located in the hospital's premise. However, we are missing one important part and that is the piece of hardware which provides the radiofrequency interface for communicating with the sensors in the WBSN. Also this module, which will be connected through the USB socket of the Raspberry Pi 2, should have a specific firmware that will be able to implement the communication protocol between the IEEE 802.15.4 and the 6lbr Linux process running in the background.

To provide the requirements above, we used one of the SensorTags as a radio interface. In this SensorTag we will flash the firmware needed for the communication protocol previously mentioned. The firmware used was a Contiki example with the name of *slip_radio.c* which uses a serial-based interface for IP packets received/sent.

Now once we configured the 6lbr application with the *slip_radio* program flashed in our SensorTag, we are ready to deploy the application by executing the command `sudo service 6lbr start`. This command will generate a file, with the name of *6lbr.ip6*, which has inside the Internet address at which the 6lbr application can be accessed. The default address will be [bbbb::100] and it will be written in our PC browser in order to access the webserver deployed in the Raspberry Pi 2.

At this point we now have a border router application being deployed in our Raspberry Pi 2 as well as a radiofrequency interface capable of communicating with the WBSN. Now it is time to try this set up and see if we are able to detect the SensorTags of what it will be our WBSN. However, before performing the test, it has to be said that for the 6lbr to be able to detect the SensorTags adhered to the patient's body, these have to be flashed with the *c26xx-web-demo* application. This is because, within this program, a 6lbr client is deployed allowing for its consequent detection.

To see if the SensorTags from the WBSN are detected, we access the webserver deployed by the 6lbr application through its Ethernet address. From figure 8, we can see that a sensor is visible on the webpage provided. Another test that was done in order to confirm connectivity with the SensorTags was performing a ping from the Raspberry Pi 2 to the IPv6 address of the SensorTag. This test was proofed to be successful as well.



The screenshot shows the 6LBR web interface. At the top, there's a header with the title '6LBR 6Lowpan Border Router'. Below it, there's a navigation menu with tabs: System, Sensors (selected), Status, Configuration, Statistics, and Administration. Under the 'Sensors' tab, there's a sub-menu with: Sensors, Node tree, PRR, Parent switch, Hop count, Traffic, and Export. The main content area is titled 'Sensors' and contains a 'Sensors list' table.

Node	Type	Web	Coap	Parent	Up PRR	Down PRR	Last seen	Status
fd00::212:4b00:615:a86b	TI	web	coap	fe80::212:4b00:615:a904	100.0%	100.0%	1	OK

Figure 8: Webpage Provided by 6lbr

About this approach to configure the WBSN, the main challenges focused in compatibility issues. Due to the continuous updates of the 6lbr application, not all versions of the software packages were fully functional. Our first installation of the 6lbr application resulted in the website not being able to access some of the tabs. Therefore, in order to solve this issue, we had to first make sure that the problem resided in the compatibility and not with our set up and then download different versions until the correct one was found.

This concludes the first alternative in configuring a WBSN. As we can see, the main idea consists in configuring the Raspberry Pi 2 as a border router between the WBSN and the outside networks.

Radio Interface through OpenLabs Module

As mentioned in the *Background* section, being able to create a gateway using the OpenLabs module as a 6LoWPAN transceiver is not as easy as installing and deploying a robust application such as 6lbr. The main reason behind this is that there is scarce work done regarding this setup throughout the sources. However, despite this inconvenience, the advantages of using the OpenLabs module are several such as its low price, its efficiency and its greater value for the research community (there is no recorded documentation in configuring this setup).

This alternative will require two important steps. The first one involves recompiling the Linux kernel of our Raspbian distribution in order for the Raspberry Pi 2 to detect the presence of the OpenLabs module in its GPIO pins. After this, we will use two Raspberry Pi 2s, with their corresponding OpenLabs module, and write a client and server application to validate that the communication is done through this installed module. The second step, which has been the most challenging one, will comprehend the proper detection of the SensorTags by the Raspberry Pi 2 through this module.

Recompiling of Linux Kernel for OpenLabs Module Operation

In this part, we will describe the process followed in order to recompile properly the Linux Kernel of our Raspbian distribution [17]. It has to be said that the building process of the kernel will be based on a cross-compilation scheme (we will build the kernel in a PC environment for its consequent deployment in the Raspberry Pi 2).

First of all, we needed to get a Raspbian image (in our case the Lite version) to flash in the SD card and which will constitute the operating system of our Raspberry Pi 2. In our case, we used the Raspbian Lite version: 2017-02-16-raspbian-jessie-lite, due to the fact that it is lighter in size and does not provide a GUI Linux (not needed for our research project). The

exact download link of this Raspbian Lite version can be viewed in the *References* section [18].

Once downloaded, we will have to use two software tools in order to format and burn the image in the SDCard correctly. Note that these tools have been used in the Windows environment:

- **SDFormatter**: software used to guarantee that there is not any type of information preexisting in our SD card.
- **Win32 Disk Imager**: program used to burn the .iso image of the Raspbian Lite distribution. When the burning process of the .iso is done, the result is an operating system in our SD card.

After these two steps we can say that we have an operative Raspbian Lite ready to boot in our Raspberry Pi 2. We plugged in the SD card and turned on our Raspberry Pi 2 and, once logged in with the default credentials (*username: pi & password: raspberry*), we typed the following command presented below

```
sudo raspi-config
```

in order to open the Raspbian Lite configuration. Once we were inside the menu, we went to **advanced options** and pressed **Expand Filesystem**. This process is done for the Raspbian Lite to utilize the whole disk space available in the SD card, in our case the whole 8 Gbytes. Once expanded the filesystem we rebooted the Raspberry Pi 2 and updated the system by entering

```
sudo apt-get update  
sudo apt-get upgrade.
```

At this point, the Raspbian Lite image is now ready to be configured with a recompiled Linux Kernel of our own to detect the OpenLabs module located in the GPIO pins.

In order to carry out this step we first needed to get a proper cross compiler for building this new kernel. In this case we used the *gcc-linaro-4.9-2014.11-x86_64_arm-linux-gnueabi*. This package was downloaded in a 64-bit Ubuntu Virtual Machine; virtual machine in which we will perform all of the needed configuration and compilation of our Linux Kernel. Once the cross-compiler tool was downloaded, we appended the following line to the PATH variable of our console.

```
export PATH=$PATH:/home/performance/Downloads/gcc-linaro-4.9-2014.11-x86_64_arm-  
linux-gnueabi/bin
```

(Note that the part in red is basically the path where our bin folder of the gcc cross compiler is).

Now once included in the PATH variable and checked that the terminal is aware of this compiling tool (by simply writing *arm-linux-gnueabi-gcc -version*), we downloaded the mutilib support by typing the following command.

```
sudo apt-get install i386.
```

This process is only necessary if the Virtual Machine used is 64-bits. Once downloaded this support, we typed the following commands in order to get all of the kernel sources that will ultimately make up our 6LoWPAN compatible kernel.

```
git clone -depth 1 https://github.com/raspberrypi/linux.git linux-rpi
```

```
cd linux-rpi
git checkout rpi-4.1.y
cd ..
```

We also had to acquire from the repository the latest firmware files of the Raspbian Lite Kernel.

```
git clone -depth 1 https://github.com/raspberrypi/firmware.git firmware
cd firmware
git checkout next
cd ..
```

With the downloaded files in our Virtual Machine, we were ready to make the correct configurations in our kernel files. Below we will explain the process that we followed in order to configure it correctly.

First of all, since we will be using a Raspberry Pi 2, we will need to work on the .iso image file named kernel7. Therefore it is important to type in the console window the following command in order to make sure that all the *#defines* referred to *KERNEL* have the correct kernel7 value.

```
export KERNEL=kernel7
```

Now in order to get a default configuration of the kernel to work on, we entered the folder *linux-rpi* where all of the kernel files were store and typed the following command.

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig
```

After acquiring this default configuration of the kernel, we needed to modify its device tree in order for the OpenLabs module to be detected from the GPIO pins. Thus, we went into the directory inside the *linux-rpi* folder: */arch/arm/boot/dts/bcm2709-rpi-2-b.dts* and we appended the following code to the file.

```
&spi0 {
    status="okay";
    spidev@0{
        status = "disabled";
    };
    spidev@1{
        status = "disabled";
    };
    at86rf233@0 {
        compatible = "atmel,at86rf233";
        reg = <0>;
        interrupts = <23 4>;
        interrupt-parent = <&gpio>;
        reset-gpio = <&gpio 24 1>;
        sleep-gpio = <&gpio 25 1>;
        spi-max-frequency = <3000000>;
        xtal-trim = /bits/ 8 <0x0F>;
    };
};
```

Once modified the device tree of our Linux Kernel, we went back to the console and opened the kernel configuration menu by first downloading the necessary packages

```
sudo apt-get install libncurses5-dev
```

and then typing

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

At this point we were prompted with a menu-like window. We then went to...

```
Networking support
--> Networking options
--> IEEE Std 802.15.4 Low-Rate Wireless Personal Area Networks support
```

and enabled this last item by a <M>. Inside this directory, we selected all the options, using again the <M>, and went back to activate the **6LoWPAN Support** option in the **Networking options**.

The previous configurations enables all the software modules necessary to work with protocols that are required in our system. However, we have yet not made available any of the IEEE 802.15.4 drivers needed for detecting the OpenLabs device. Therefore, we went to the following directory in the menu.

```
Device Drivers
--> Network device support
--> IEEE 802.15.4 drivers
```

and again activated this last part with a <M> as well as all of the options within this directory. Returning to the main menu, we also went to **Boot Options** and made sure that the following line

```
console=ttyAMA0,115200 kgdboc=ttyAMA0,115200 root=/dev/mmcblk0p2 rootfstype=ext4
rootwait
```

was written in the **() Default kernel command string**. This concludes configuring all the necessary options for our Linux Kernel to detect and utilize the OpenLabs module for IEEE 802.15.4.

The following step is just building the configured kernel image by entering the following command.

```
CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm make zImage modules dtbs -j4
```

With the kernel image built, we overwrite the kernel files stored in our SD card through the following commands.

```
cd /home/performance/linux-rpi
sudo cp arch/arm/boot/dts/*.dtb /tmp/mnt/boot
sudo cp arch/arm/boot/dts/overlays/*.dtb* /tmp/mnt/boot/overlays
sudo scripts/mkknling arch/arm/boot/zImage /tmp/mnt/boot/$KERNEL.img
```

We also install the kernel modules, which were selected in the previous mentioned configuration menu, in the root partition of the SD card, along with the new firmware files.

```
sudo CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm INSTALL_MOD_PATH=/tmp/mnt/root
make modules_install
cd ..
cd firmware
sudo rm -rf /tmp/mnt/root/opt/vc
sudo cp -r hardfp/opt/* /tmp/mnt/root/opt
```

After all of this, we opened the config.txt file located in the directory of **boot** in our SDCard and wrote the following line right below the **#dtoverlay=lirc-rpi** for the system to finally detect the OpenLabs device.

```
dtoverlay=at86rf233
```

After this process, our Raspberry Pi 2 will be able to detect the OpenLabs module located in its GPIO pins. However, we will need to install a software package known as *wpan-tools* in the Raspberry Pi 2 to be able to configure a 6LoWPAN network interface. The *wpan-tools* is a set of tools which implement the IEEE 802.15.4 stack and therefore will be necessary if we want to configure a 6LoWPAN network. The below commands were used in order to install and set up this software package, along with its dependencies.

```
sudo apt-get install git libnl-3-dev libnl-genl-3-dev dh-autoreconf
git clone https://github.com/linux-wpan/wpan-tools
cd wpan-tools

--Configuring options for the wpan-tools
./autogen.sh
./configure CFLAGS='-g -O0' --prefix=/usr --sysconfdir=/etc --libdir=/usr/lib

--Compilation/installation of the tools
make
sudo make install
```

Now we were ready to establish a 6LoWPAN interface in our Raspberry Pi 2. Using the *wpan-tools* just downloaded we generated a script that will automatically launch at start-up and will configure the interface wanted. The script can be seen below.

```
--Set PAN ID to 0xacdc
sudo iwpan dev wpan0 set pan_id 0xacdc

--Configure the 6LoWPAN interface and activate it
sudo ip link add link wpan0 name lowpan0 type lowpan
sudo ifconfig wpan0 up
sudo ifconfig lowpan0 up

--Erase previous contents of LowPAN address and configure statically a new one.
sudo ip addr flush dev lowpan0
sudo ip addr add fe80::212:4b00:6a0:a400/64 dev lowpan0
```

Once we executed the script generated, we had a Raspberry Pi 2 with a 6LoWPAN interface and its consequent IPv6 address. From the figure below we are able to see the new network interface created with the name of *lowpan0*.

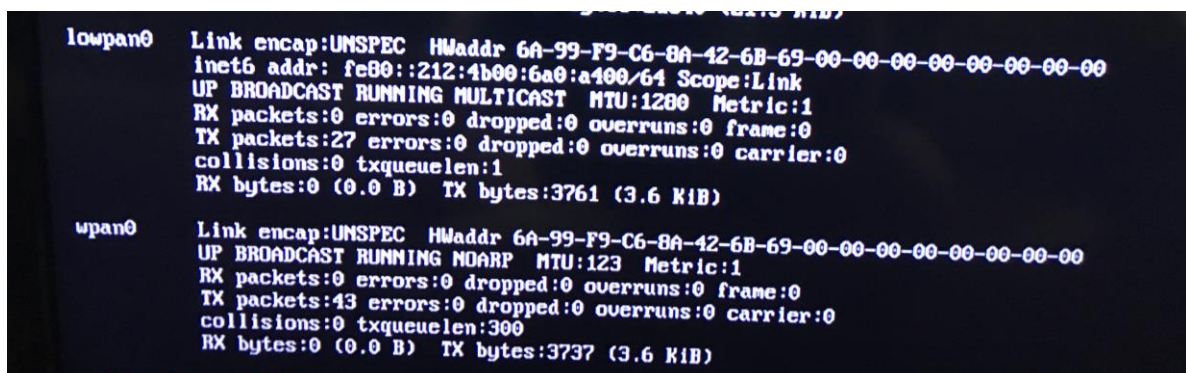
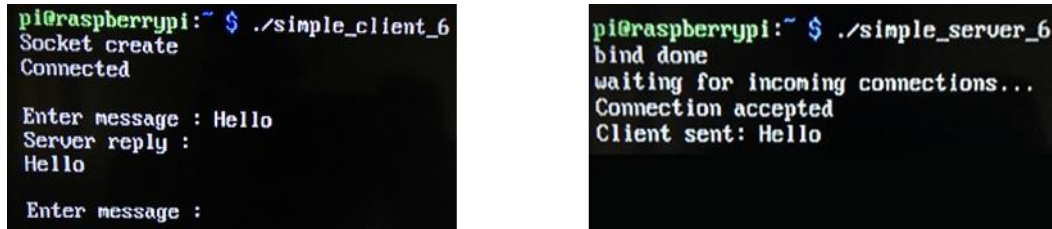


Figure 9: 6LoWPAN Network Interface in the Raspberry Pi 2

Now in order to verify that the interface was functioning correctly, we repeated the whole

process above mentioned in a second Raspberry Pi 2, only this time in the start-up script we changed the address of the 6LoWPAN to fe80::212:4b00:6a0:a405. We then created two programs, a server and client written in C, whose main purpose will be to use these IPv6 addresses generated for communication. The below images show the success in communicating the two Raspberry Pi's through the 6LoWPAN interface created.



```

pi@raspberrypi:~$ ./simple_client_6
Socket create
Connected

Enter message : Hello
Server reply :
Hello

Enter message :

pi@raspberrypi:~$ ./simple_server_6
bind done
waiting for incoming connections...
Connection accepted
Client sent: Hello

```

Figure 10: (left) Client and (right) Server Deployed through 6LoWPAN

As a last test, in order to verify that this was done through the 6LoWPAN interface provided by the OpenLabs module, we removed in the middle of the communication the two modules. It was seen that if taken out, the communication failed. Therefore, we can safely say that the two Raspberry Pi 2s were communicating through the 6LoWPAN module.

This concludes the procedures done regarding the installation and operation of the OpenLabs module.

SensorTag Detection through the OpenLabs Module

At this point, our Raspberry Pi 2 is able to communicate via 6LoWPAN thanks to the OpenLabs module that has been installed. It is now time to, at least, be able to detect the sensors that will make up the WBSN. It has to be said that this task has been one of the most difficult challenges since there is no documentation recorded regarding this issue.

Before going into the actual technical work performed to manage the communication between the Raspberry Pi 2 and the SensorTags, we need to take a look at figure 9. As we can see, the address given to the *lowpan0* interface pertains to the link-local address block (fe80::/64). The use of this set of addresses implies that the devices are only able to communicate if they belong to the same broadcast domain. If we want the SensorTags to communicate with the Raspberry Pi 2 recently created network interface, we will need for these to also have addresses that belong to this link-local address block. Therefore, networking communications will be done with addresses of the type fe80::/64.

Now in order to verify that our gateway has visibility on the SensorTags, we will be using the *ping6* command from our Raspberry Pi 2. As we know, the ping command obviously requires an IPv6 address into which send the ICMP packet and in this part it is where our first challenge arose. From the *Background* section of this report, we mentioned that the communication through the OpenLabs module did not require the 6lbr application and was therefore uninstalled. One of the benefits of this application was that we were able to see the IPv6 addresses of our SensorTags through its webserver deployed. Without it, we can't know the IPv6 of our sensors and therefore we don't know to which IPv6 address we should send the ICMP packets to check the visibility between our gateway and the sensors.

In order to overcome this problem, we flashed a USB dongle device with a *sniffer.c* program. The flashed USB was connected to the PC and Wireshark was launched in order to view the packets. From the sniffed packets, we could see that the SensorTags periodically sent messages (neighbor advertisements) to the medium and viewing the source address in

Wireshark we could get the IPv6 of the device.

Despite acquiring correctly the IPv6 of our sensors, when our gateway performed a ping to one of them, there was no response. In Wireshark we could perfectly see that the packet was sent however, the SensorTag was not able to receive it and answer back to the Raspberry Pi 2. After some research it was found that, in order for the SensorTags to perform the network discovery, the RPL protocol is used. When implementing the 6lbr solution, the RPL is already built and that is why the SensorTags are able to connect successfully to the Raspberry Pi 2. However in Linux stand-alone, the approach we are using right now with the OpenLabs module, this protocol is not implemented. Therefore, in the source code of the SensorTags (*contiki-conf.h*), we had to add the following lines.

```
#undef UIP_CONF_IPV6_RPL
#define UIP_CONF_IPV6_RPL 0
```

After these modifications, the Raspberry Pi 2 was still not able to receive the answer back from the SensorTag to its ping request. For successfully achieving the visibility of the SensorTags by the Raspberry Pi 2, we needed to modify these following lines in the same file mentioned previously

```
#ifndef NETSTACK_CONF_MAC
#define NETSTACK_CONF_MAC csma_driver

#ifndef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC contikimac_driver
```

to

```
#ifndef NETSTACK_CONF_MAC
#define NETSTACK_CONF_MAC nullmac_driver

#ifndef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC nullrdc_driver
```

These lines modify Contiki's network stack whose architecture can be seen in the figure on the right. Basically, the Radio Duty-Cycle (RDC) layer is responsible for setting the sleep periods in the sensor nodes. Choosing the type of RDC layer is really important since it determines when the packets will be sent as well as it makes sure that the node is functioning when it receives the packets. On the other hand, the MAC layer is the well-known Media Access Control layer responsible for the channel access control. Therefore, the changes commented earlier are basically simpler implementations of each of the two layers which are compatible with the Linux operating system in hands.

Once done all of the changes commented, we performed a ping from the Raspberry Pi 2 to the IPv6 address of one of the sensors and a reply was received from the SensorTag, as it can be seen in the figure below. Therefore we can say that communication is possible between the Raspberry Pi 2 and the SensorTag through the OpenLabs module.



Figure 11: Layer Organization in ContikiOS

```

pi@raspberrypi:~$ ping6 fe80::212:4b00:6a0:a402%lowpan0
PING fe80::212:4b00:6a0:a402%lowpan0(fe80::212:4b00:6a0:a402) 56 data bytes
64 bytes from fe80::212:4b00:6a0:a402: icmp_seq=1 ttl=64 time=8.82 ns
64 bytes from fe80::212:4b00:6a0:a402: icmp_seq=2 ttl=64 time=8.73 ns
64 bytes from fe80::212:4b00:6a0:a402: icmp_seq=3 ttl=64 time=10.4 ns
64 bytes from fe80::212:4b00:6a0:a402: icmp_seq=4 ttl=64 time=10.3 ns
^C
--- fe80::212:4b00:6a0:a402%lowpan0 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3003ms
rtt min/avg/max/ndev = 8.739/9.585/10.406/0.809 ms
pi@raspberrypi:~$

```

Figure 12: Ping Request to SensorTag via the OpenLabs Module

It has to be said that, from now on, in order to discover the IPv6 addresses of all the sensor nodes we will execute the following command in the Raspberry Pi 2.

```
ping6 ff02::1%lowpan0
```

This command broadcasts packets through the *lowpan0* interface. Therefore, all the SensorTags present in the same medium link will respond to these ping packets sent by the Raspberry Pi 2. The terminal window, when executed this command, is shown in figure 13 where it can be clearly seen how the addresses of the SensorTags present in the medium are shown (squared in red). This way we don't need WireShark anymore to sniff the packets present in the medium in order to retrieve their addresses.

```

pi@raspberrypi:~$ ping6 ff02::1%lowpan0
PING ff02::1%lowpan0(ff02::1) 56 data bytes
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=1 ttl=64 time=0.224 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=2 ttl=64 time=0.150 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=3 ttl=64 time=0.145 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=4 ttl=64 time=0.123 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=5 ttl=64 time=0.100 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=6 ttl=64 time=0.120 ms
64 bytes from fe80::212:4b00:6a0:a402: icmp_seq=6 ttl=64 time=9.55 ms (DUPT)
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=7 ttl=64 time=0.166 ms
64 bytes from fe80::212:4b00:6a0:a402: icmp_seq=7 ttl=64 time=10.2 ms (DUPT)
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=8 ttl=64 time=0.159 ms
64 bytes from fe80::212:4b00:6a0:a402: icmp_seq=8 ttl=64 time=9.60 ms (DUPT)
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=9 ttl=64 time=0.122 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=10 ttl=64 time=0.115 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=11 ttl=64 time=0.118 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=12 ttl=64 time=0.114 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=13 ttl=64 time=0.165 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=14 ttl=64 time=0.124 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=15 ttl=64 time=0.117 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=16 ttl=64 time=0.215 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=17 ttl=64 time=0.131 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=18 ttl=64 time=0.140 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=19 ttl=64 time=0.116 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=20 ttl=64 time=0.100 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=21 ttl=64 time=0.110 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=22 ttl=64 time=0.110 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=22 ttl=64 time=10.9 ms (DUPT)
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=23 ttl=64 time=0.113 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=23 ttl=64 time=9.59 ms (DUPT)
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=24 ttl=64 time=0.116 ms
64 bytes from fe80::212:4b00:6a0:a400: icmp_seq=24 ttl=64 time=9.51 ms (DUPT)
^C

```

Figure 13: ping6 ff02::1%lowpan0 response from two SensorTags

From the results given in this section, we can guarantee that our gateway now has a 6LoWPAN network interface deployed and it is able to detect and communicate, from this interface, with the sensors that will be located in the WBSN.

At this point we then have two alternatives for deploying the WBSN: deploying a border router with the 6lbr application and a SensorTag as the transceiver module or a unique and own solution in which the Raspberry Pi 2 acts as a gateway and uses the OpenLabs module as a transceiver. Once studied the advantages and disadvantages of each implementation, explained in the *Background* section, we decided to use for our WBSN the latter alternative. Therefore from now on, when we refer to the WBSN in this report, we will keep in mind that the network architecture is the one made with the OpenLabs module.

Operating the WBSN

From the completion of the above objective, our gateway is able to communicate with any of the SensorTags of our WBSN. What we need to do now is create a program of our own that will be able to not just check the presence of the sensors but also request data from them.

In order to do so, the approach chosen is to deploy on the SensorTags a CoAP server and on the Raspberry Pi 2 a CoAP client. CoAP, as mentioned earlier, is based on a REST model which basically uses methods such as GET or POST in order to interact with other nodes. Therefore, the Raspberry Pi 2 will send GET requests to the sensors in order for these ones to send the data asked back. Now in order to work with the CoAP protocol in our Raspberry Pi 2, we will use the libcoap library [19]. This library is a C-implementation of CoAP that provides the functions needed for operating with this protocol.

In a general manner, in order to request information using libcoap from a CoAP client (the SensorTag), we will need in our CoAP server to establish the destination address to which we will want to send the CoAP requests. Also we will need to create a *coap_context_t* structure which stores the CoAP stack's global state. The following code illustrates the initial setup for sending requests to a CoAP client.

```
//Specification of one sensor for CoAP protocol communication (context1)
//Determines from string both uri.host and uri.port
cmdline_uri("coap://[fe80::212:4b00:6a0:a402%lowpan0]:5683");

//Assigns to variables the host and port number for then...
server1 = uri.host;
port = uri.port;

//Resolving its address
result = resolve_address(&server1, &dst1.addr.sa);

if (result < 0) {
    fprintf(stderr, "failed to resolve address\n");
    exit(-1);
}

dst1.size = result;
dst1.addr.sin.sin_port = htons(port);

//Creates a CoAP context in which the CoAP stack's state will be stored
ctx1 = get_context("::", port_str);

if (!ctx1) {
    printf("Can't create context");
    return -1;
}
//Assigns to the given context an specific message handler function
coap_register_response_handler(ctx1, message_handler);
```

From the previous code we can see that the initial set up to register a sensor is straightforward; it looks similar to when working with network sockets. The only line which might seem troublesome to understand is the operation of the *coap_register_response_handler*. Basically what this line does is specify to what function

should the program head to when it receives data in the socket of the `coap_context` structure. We need to take into account that the data sent from the SensorTag to the Raspberry Pi 2 is in text format, therefore the `message_handler` function should perform the mapping from a text to a number representation.

If we were to add another sensor to our WBSN architecture, we will need to perform the same set of instructions again only this time changing the IPv6 address inside the `cmdline_uri` function. This way we then have two contexts, one for each of the SensorTags, and two different `dst` structures. These variables will be used to send the CoAP requests to the SensorTags for retrieving the different sensor magnitudes. Below we can see the set of C instructions written in order to fetch the x-axis accelerometer value from one of the sensors.

```
// Sets the resource it wants to fetch "sen/mpu/acc/x"
cmdline_uri("coap://[fe80::212:4b00:6a0:a402%lowpan0]:5683/sen/mpu/acc/x");

// Creates a PDU with the request it will send
pdu1 = coap_new_request(ctx1, 1, optlist);

// It sends a confirmed CoAP message to a destination (needed for GET requests)
coap_send_confirmed(ctx1, &dst1, pdu1);

// Clears the file descriptor used for reading and then assigns the file descriptor to the socket
// associated to the CoAP context.
FD_ZERO(&readfds1);
FD_CLR( ctx1->sockfd, &readfds1);
FD_SET( ctx1->sockfd, &readfds1);

// Monitors the file descriptor to check if it's ready for reading (a new value has arrived). Also a
// timeout has been set.
result=select(ctx1->sockfd+1,&readfds1,0,0,&tv1);
if(result==-1) arm_error_x=1;

// Reads the data from the network parsing it as a CoAP PDU
result=coap_read(ctx1);
if(result==-1) arm_error_x=1;

// Dispatches any PDUs present in the receiving queue.
coap_dispatch(ctx1);
if(value>2) arm_error_x=1;

// Information processed (irrelevant)
if(arm_error_x!=1){
...
}
```

At the `coap_read(ctx1)` line, the `message_handler` is called in order to process the PDU received and convert the value sent by the SensorTag from text into a float variable denoted as `value`.

The process explained above is done for extracting one magnitude of a single sensor. Now, if we wanted to read other magnitudes, we will have to copy and paste the above lines of code changing, once again, the string in the `cmdline_uri` function. This way we can extract different values from the different sensors present in the multiple SensorTags. The idea will then be to have a while (1) loop in which, in every iteration, we will request a different magnitude from the sensors.

While implementing this application, it was seen that if we were to send right away in each iteration many different requests, one after the other, there will be a higher chance of losing in transmission one of the data packets. Since our CoAP client waits until it gets a reply from a CoAP server, if the packet is lost in transmission then the program will be waiting forever. The reason behind this packet loss is due to the nature of the SensorTags; they are really cheap devices which often tend to fail when they are overworked. Other reasons might be due to the coverage of the SensorTag antenna, the interferences existing in the medium, etc....

The first approach in solving this problem is to establish a specific scheduling algorithm, deciding when each of the SensorTag's magnitude will be requested. In order to do so, we assigned a counter variable to each SensorTag's magnitude and also we established a sleep period between iterations in the while (1) loop of 1 ms. In each loop iteration, every counter variable will increase in one unit and these counters, when they reached a certain value, will send a request to the SensorTag asking for a specific magnitude. In order to specify the size of each counter, we took into consideration the refreshing rate needed for that magnitude in the application. If a more real time refreshment was needed, a smaller counter value will be used, however if the update requirement was not strict the counter size will be large. This way there is not an excessive number of packets exchanged through the medium and a lower chance of losing one.

Although we reduced the use of bandwidth in the communication, there was still times in which the response to a request didn't reach the Raspberry Pi 2 correctly, hanging once again, the program. It was seen that our software will wait indefinitely in the *coap_read* function. Therefore we had to modify this function present in the libcoap library in a way that will let us specify a timeout value. This way, if the response didn't arrive within a time interval, we will forget about that request and carry on with the program. Below we can see the modification that was made in the *coap_read* from

```
bytes_read = recvfrom(ctx->sockfd, buf, sizeof(buf), 0, &src.addr.sa, &src.size);
```

to

```
//If condition that waits for a time with a timeout for the socket to receive data (==0 → received data)
```

```
if(setsockopt(ctx->sockfd, SOL_SOCKET, SO_RCVTIMEO, &read_timeout,
sizeof(read_timeout))==0)
```

```
    bytes_read = recvfrom(ctx->sockfd, buf, sizeof(buf), 0, &src.addr.sa,
&src.size);
```

Through this solution, while it is true that we have not improved the success rate of responses to the Raspberry Pi 2, we at least have reduced the impact of the errors: now the program will carry on with other requests and not hang indefinitely.

This concludes the operation of our WBSN: the Raspberry Pi 2, CoAP client, will request the SensorTags, CoAP servers, for the appropriate sensor data. Once the data reaches the Raspberry Pi 2, we will be able to process it accordingly in order to derive the state and condition of the patient.

Deploy Processing Algorithms in the Raspberry Pi 2

From the work previously described, we have a system which is able to transmit the sensor

data to a processing gateway, in this case the Raspberry Pi 2. Now, the processing power of the Raspberry Pi 2 is very high and therefore it is interesting to perform some type of processing in the data acquired from the nodes rather than just send the raw information directly to the server database. The objective will be to handle this data in a way that we could derive information regarding the state and condition of the patient. This section will describe the different applications our health monitoring system will be able to provide. It has to be said that we will save for the *Results* section the actual screenshots of the different functionalities working correctly.

Position Tracking of the Patient

One interesting application to deploy in our health monitoring system is position tracking. The major benefits of this application is detecting where the patient is at a given time in case something wrong has happened to him/her. This way specialists can move right away to the location the patient is without losing time in his/her search.

In order to implement this functionality, we thought in requesting the three axis of the accelerometer data from the Raspberry Pi 2 given by the MPU9250 component present in the SensorTags [20] [21]. The values of these components are in m/s^2 and therefore, it looked at first, as if by simply integrating twice this data we will acquire the position of the person at a precise moment. However after some research, it was seen that the SensorTags that we are dealing with are not as precise as expected. Also, since we are integrating the accelerometer components, we are introducing an error known as the integration error which accumulates over time. This error is far worse when integrating twice and therefore an alternative had to be thought in order to implement this functionality.

The approach taken was that of a pedometer device. Basically what we will want to do is measure the number of steps that it is taken by a patient to know roughly the distance he/she has travelled (assume each step is one meter). Thus, if the patient has walked 10 steps, he/she will have travelled 10 meters. Now, in order to detect the position in a given X Y plain, we will also need to know what direction he/she is facing. The program will start and take as a reference the direction he/she is looking and the actual position he/she is on. Every time he/she turns we will modify the value of this angle α and, knowing the distance he/she has travelled, we can make an estimation of where he/she is positioned. The formulas used to determine the X and Y coordinates are shown below.

$$\begin{aligned} position &= (X, Y) \\ X &= steps * \sin(\alpha) \\ Y &= steps * \cos(\alpha) \end{aligned}$$

In order to detect angle measurements, we will need to use the gyroscope data present in the MPU9250 of the SensorTag. The value provided by this component is in degrees/sec and therefore we will need to integrate it once in order to acquire actual degrees. It is true that before we said that integrating gives an error too hard to handle, however this time we are only integrating once and thus we might be able to cope with the integration error problem. With this new approach, we will be able to give a rough estimation of the position of the patient without the need of high power consumption alternatives such as GPS signals.

To track the position, we first had to establish several things such as how many sensors we will need, where will we place them and what axis components of both the accelerometer and gyroscope we will use. For the first question, we thought that it should be necessary at

least two SensorTags, one that will detect the angle we are facing and the other one that will focus on detecting steps. After deciding that one of the SensorTags will focus in counting steps, we placed this one in different parts of the body (wrist, ankle, leg, etc...). We then evaluated the correlation between the accelerometer data extracted from each part of the body with the patient performing the actual step.

It was seen that when the accelerometer was placed in the wrist, we could intuitively discern when a step occurred through the variations of the values in the y-component. On the other hand, for the direction that the patient is facing, we could see that the best option was to place the second SensorTag on the patient's chest and evaluate the gyroscope's x-component. In the figure below we can see, on the left, the placement of the two SensorTags in the patient's body and, on the right, the axis' representation in the SensorTag.

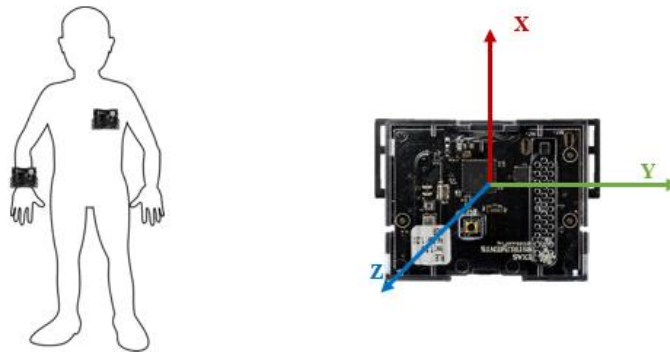


Figure 14: (left) Placement of Sensors in Patient (right) Definition of Axis in SensorTag

Once established the placement of the SensorTags and the axis components to process of both accelerometer and gyroscope, it is time to model the y-axis accelerometer variation into a C program in order to detect when a step has occurred. In order to do this, several sets of data were drawn from the arm sensor while walking and have been plotted in Matlab. In figure 15 we give three plot samples of the variation of the y-axis accelerometer when the patient is walking. Note that we illustrated two concepts that will be commonly used in this part of the report: valley and peak.

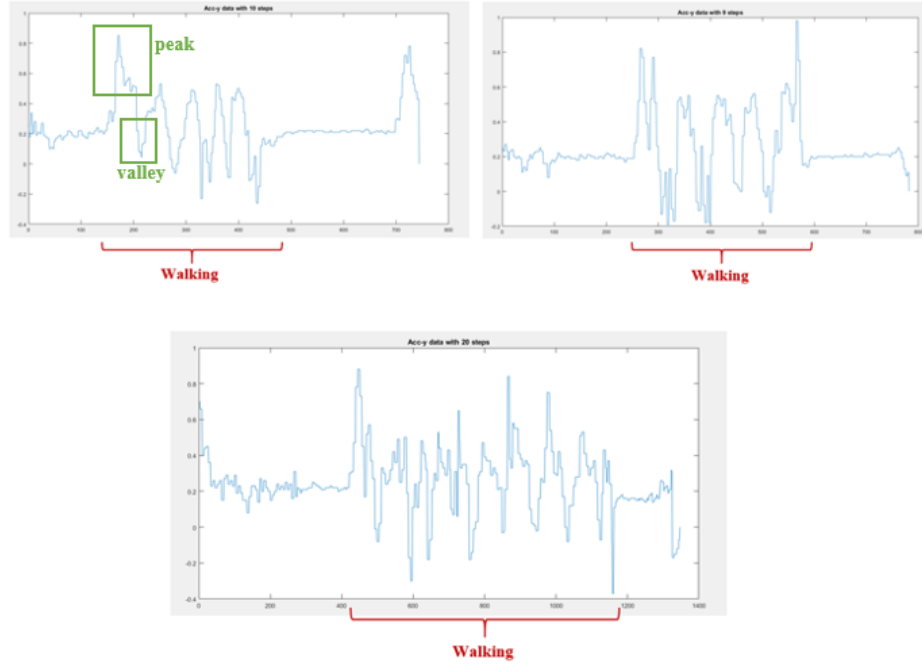


Figure 15: Walking sets of data: (top left) 10 steps, (top right) 9 steps, (bottom) 20 steps

From the graphs above we can see a certain trend: the number of peaks and valleys are closely related to the number of steps. The peaks occur when the person moves their left arm upwards, arm in which the sensor is placed, when taking a step with the right leg. On the other hand, the valley occurs when the left arm is moved downwards when a step is made with the left leg. However when trying to model this behavior into our CoAP server program in the Raspberry Pi 2, we encountered several challenges.

Our approach in detecting these peaks and valleys was searching for maximums and minimums that were above or below a certain threshold from the rest value. Due to the presence of high frequency noise coming from both our movement and the SensorTag's readings, some of these sporadic noise samples could be mistakenly taken as steps reducing greatly the precision of our system. Therefore, we needed some sort of tool that will smooth the data requested from the sensors in order to discern better the presence of steps during the walking action.

For this we used a low pass filter implementation to reduce these high frequency samples. This filter was of first order and is defined by the following differential equation

$$y[n] = y[n - 1] + a * (x[n] - y[n - 1])$$

where

$$y[n] = \text{accelerometer filtered sample } n$$

$$x[n] = \text{accelerometer sample } n$$

$$a = \frac{dt}{RC + dt} \quad RC = \frac{1}{2 * \pi * f_{cutoff}}$$

From the above equation we can see that the only two unknown variables are the cutoff frequency (f_{cutoff}) and the time between samples (dt). In order to find reasonable values for these two variables we used an empirical method and decided that the values that gave an

appropriate representation for processing were a $f_{cutoff} = 20$ Hz and a $dt = 1$ ms. Below we can see the plots of the filtered accelerometer data after establishing these values.

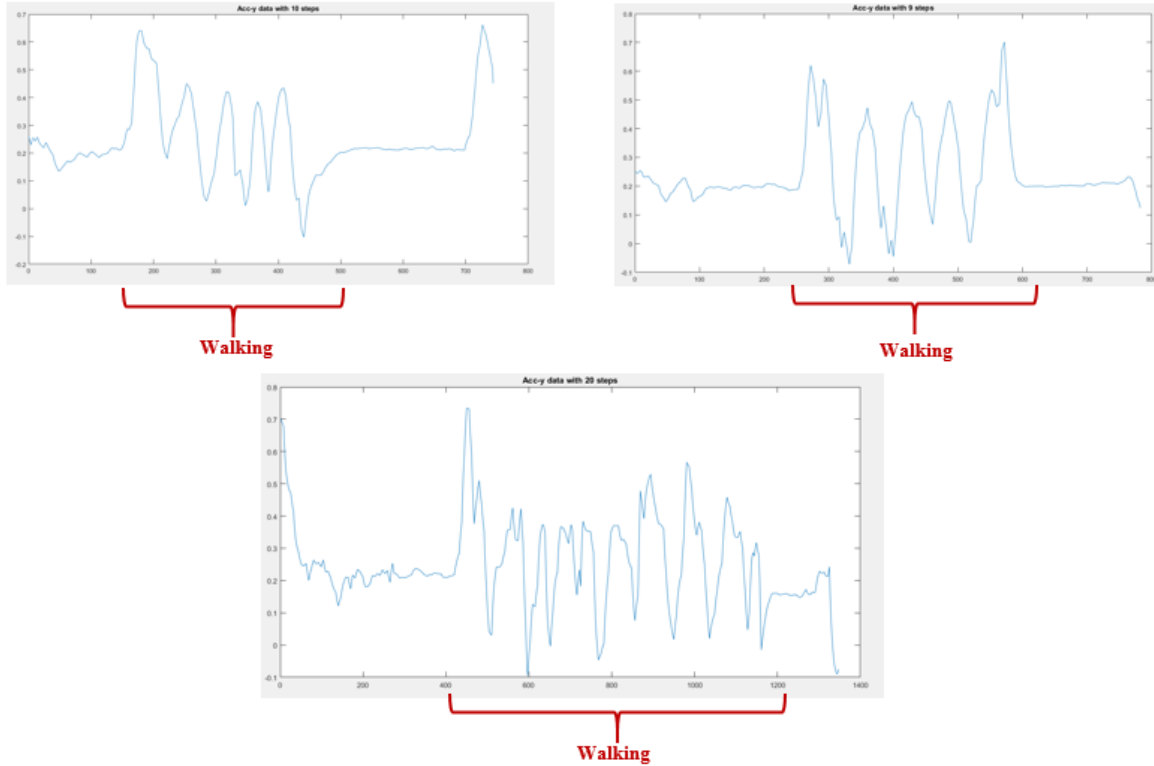


Figure 16: Walking sets of data after Filter: (top left) 10 steps, (top right) 9 steps, (bottom) 20 steps

From the three plots above, we can see that we have reduced the sporadic high frequency peaks that existed in figure 15. This way it is much easier to develop a processing section in our CoAP server which is able to extract valuable information when the patient is walking. In order to do so, we have established the following conditions in our scenario:

- The patient starts walking when the accelerometer filtered data is above a specific value. This means that the patient will always have to start walking by raising their left arm first.
- In order to detect steps, we have established that a step occurs when the difference between the previous maximum and the actual minimum (or the previous minimum and the actual maximum) is larger than a specific threshold. This way the small peaks present in the data are not taken mistakenly as steps.
- The user stops walking when the accelerometer data doesn't vary greatly over a considerable amount of samples.

With these set of conditions we were able to develop a C code that will detect when a patient has started walking, when a step has been taken and when the walking action stops. Below we give a high level description of the algorithm implemented. (Note that references to *acc_y* and *sample* denote the filtered accelerometer data).

```
if (|acc_y-acc_y_rest| > threshold_start_walking) --> user has started walking.
state_patient = walking
```



```

find = maximum                                --> we will be looking for a maximum
steps = 0

if (state_patient==walking)                   --> processing during walking

    if(find == maximum)                       --> we are looking for a maximum
        if(previous_sample > present_sample) --> possible maximum found
            difference= previous_sample - minimum
            if (difference > threshold)        --> maximum found is a step
                find=minimum                  -->we will now look for a min value
                maximum=previous_sample      --> record the maximum found
                steps ++                      --> a step has occurred

    if(find == minimum)                       --> we are looking for a minimum
        if(previous_sample < present_sample) --> possible minimum found
            difference= maximum - previous_sample
            if (difference < threshold)        --> minimum found is a step
                find=maximum                  -->we will now look for a max value
                minimum=previous_sample      --> record the minimum found
                steps ++                      --> a step has occurred

    if (|present_sample- value_at_rest|<threshold_stop_walking) --> detect rest
        samples_inactivity++
        --> if no variation from value at rest for a number of samples
        if(samples_inactivity>certain_number_samples)
            patient_state=standing

```

This algorithm will have to be implemented inside a while (1) loop in order to always evaluate when the patient has started walking, when he/she has taken a step and when he/she has reached a standing position. Obviously there is more conditions that need to be taken into consideration during the walking state, however the above are the main ones that should be understood by the reader. Further details can be found in the source code.

This concludes the processing part for detecting when a patient is walking. Now to detect their position, we needed to know the angle their facing in order to calculate the X and Y coordinates. To do this we used the gyroscope data provided by the x-axis of the SensorTag located in the chest of our patient. As mentioned earlier, the gyroscope data sends the information as degrees/s and therefore we needed to integrate this data, which basically means perform a sum between the present and previous sample.

$$\begin{aligned}
 angle[n] &= angle[n - 1] - ((gyroscope_{value}[n] - drift) * dt * stabilization) \\
 stabilization &= 0.62 \\
 dt &= 0.02 (s)
 \end{aligned}$$

Notice how, in the previous formula, we have added two parameters to calculate our present angle: the drift and the stabilization. The drift is basically a mechanical error the gyroscope has, meaning that when at rest the gyroscope will always be providing a reading different from 0. Therefore, we need to subtract this error from the present reading. On the other hand, the stabilization parameter is basically a value we had to add that was computed empirically by trials with our application. To find it, we basically turned to a certain degree of orientation, say 90°, and checked what value the stabilization parameter should be in order to give angle [n] this value. This was tried with several orientations until a proper and constant value was found.

The orientation has been modelled as an independent state of the patient, meaning that he/she is only to change his/her angle of direction when he/she is in a standing position. Even though we had taken into consideration the error that the gyroscope gives at a standing still position, there is still a small variance in the data. In order to avoid this small variation to be taken as if the patient is turning, we have added a small threshold value: if the gyroscope reading is above or below this threshold he/she is either turning clockwise or anticlockwise. Below we give the high level algorithm followed for orientation detection.

```

if (|gyro_x - drift|>threshold_start_rotation)    --> user has started rotating
    state_patient = rotating

if (state_patient==rotating)                    --> processing during rotating

    if(|gyro_x - drift|>threshold_start_rotation)    --> the user is rotating
        angle[n]=angle[n-1]-((gyro_x-drift)*dt*stabilization)
    else
        samples_stopped_rotating++

if (samples_stopped_rotating>certain_value)    --> the user has stopped rotating
    state_patient = standing

```

Once again, there are more details for detecting the angle of direction however this is just a broad description of the algorithm used.

This concludes the description of the processing algorithms implemented in order to compute an estimation of the position of the user.

Sitting and Lying Down Detection

At this point our C application is able to detect when the user is standing still, when he/she is walking and calculate the direction he/she is facing. We decided to enhance the functionality of our system by also detecting when he/she is sitting and lying down.

We will start off by explaining how we detect the sitting down state in the patient. In order to do so, we first defined the following condition in this scenario: when the patient sits down he/she rests his/her hands in his/her thighs. This means that the z-axis of the wrist accelerometer will give a value near 9.8 m/s^2 , due to the presence of gravity and the position that the SensorTag is on. In order to detect when the user is standing back up we used the value given by the x-axis of this same SensorTag and compare it to the value at standing up. The reason behind using the x-axis instead of the z-axis is that this way, we allow a higher freedom in the movement of the arm when the patient is sitting down.

So basically, from the description, we can see that we will implement an algorithm which uses threshold values to discern between these two states. Below we will give the high level algorithm that we will implement in order to detect when the user is sitting down. Note that in here the filter is not applied on the accelerometer data.

```

if (|acc_z-acc_z_rest|>threshold_sitting)    --> user might be sitting down,
increase samples
    sitting_down_samples++

if (sitting_down_samples>certain_value)        --> user is sitting down
    patient_state=sitting

if (state_patient==sitting)                    --> processing during rotating

```

```
if(|acc_x-acc_x_rest|<threshold_standing_up) --> the user might be standing
    standing_samples++
```

```
if (standing_samples > certain_value) --> the user is standing up
    state_patient = standing
```

For detecting when the user is lying down, we will use a similar procedure to the one explained above. However, in this case, we will use the accelerometer z-axis component of the SensorTag located in the chest. When the patient is either sitting down or standing up, the z-axis component doesn't perceive the influence of the acceleration of gravity, giving a value near to 0 m/s². As soon as he/she is lying down, the z-axis of the SensorTag in the chest will be aligned with the direction of gravity giving therefore a value near to this one. Therefore, in order to detect if the user is lying down, we will need to examine the value given by the z-axis accelerometer.

```
if (|acc_z-acc_z_rest|>threshold_lying) --> user might be lying down,
increase samples
    lying_down_samples++
```

```
if (lying_down_samples>certain_value) --> user is lying down
    patient_state=lying
```

```
if (state_patient==lying) --> processing during lying down
```

```
if(|acc_z-acc_z_rest|<threshold) --> the user might be sitting
    sitting_samples ++
```

```
if(|acc_x_arm-acc_x_arm|<threshold) --> the user might be standing
    standing_samples ++
```

```
if (sitting_samples > certain_value) --> user is sitting
    state_patient = sitting
```

```
if (standing_samples > certain_value) --> user is standing
    state_patient = standing
```

Notice how in this algorithm, when the accelerometer value of the chest SensorTag is not within the lying down range, the user will be assumed to be sitting down. However, by checking the accelerometer x-axis of the wrist sensor node while he/she is lying down, we will be able to detect when the user transits from a lying down position to directly a standing up without sitting down in between. If we were to neglect this last condition, when the user goes from lying down to standing directly, our application will transit from lying → sitting → standing. This sitting state is mistakenly added and therefore will not accurately describe the activities performed by the patient.

This concludes the sitting and lying down algorithms. As we can see, in order to detect these two new states we will basically reside in the accelerometer data drawn from both the wrist and chest SensorTag.

Running Detection

Another functionality we thought that our monitoring system should have is to be able to detect when the patient is running. In order to do so, we followed a similar procedure to that

of the walking detection: we gathered all the accelerometer data from the two SensorTags and analyzed which of the axis showed a greater correlation to the actual running activity. Through Matlab, it was seen that the x-axis of the sensor node located in the chest showed a variation from which we could easily infer when a step was taken during running.

In the figure below we can see the raw x-axis data acquired from the SensorTag in the chest when the user is running. In the left image the patient has ran a number of 8 steps while on the right one 13.

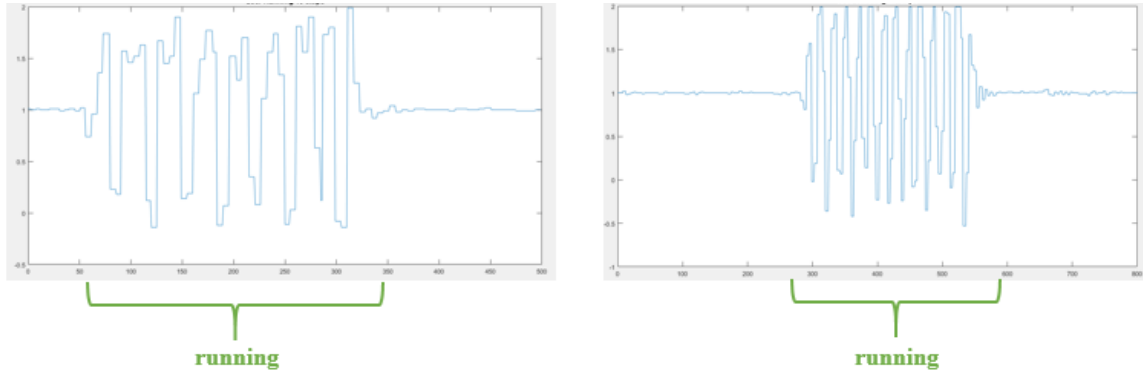


Figure 17: User Data Ran for (left) 8 steps (right) 13 steps

From the above figures, we can see that the plots, once again, are not smooth as we will want them to be. If we were to smooth them, our application will be able to discern better the steps taken and thus be more precise. In order to do so, we implement the same low pass filter used for the walking detection part, along with the same parameters. In figure 18 we show the result once the raw data from the SensorTag has gone through the low pass filter.

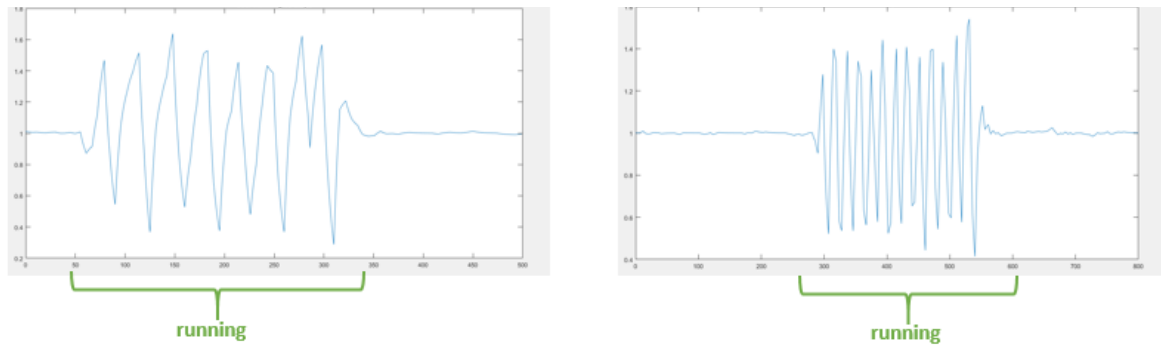


Figure 18: User Filtered Data Ran for (left) 8 steps (right) 13 steps

We can see in the figure above how the peaks and valleys are more noticeable when the filter is applied. At this point, we are now ready to implement the C processing section that will be able to detect when the user starts running, when he/she performs a step and when he/she stops running. In order to do so, we must take into account that the conditions will not be as complex as with the walking state. In this case there are no small maximums and minimum which might get mistakenly counted as steps and therefore the implementation will be a bit simpler. However it has to be said that, at an algorithmic level, the approach for detecting steps is very similar.

From figure 18, when the user starts running there is a decrease in the accelerometer x-axis value. Therefore, we will implement a condition that evaluates whether the

accelerometer x-axis is below a certain threshold. If it is then, the user has started running. When the user starts running we will firstly search for a maximum value. At this point it is when there is a difference in comparison to the walking detection algorithm. We will look for a maximum sample and, once it is found, we will save it in a variable and start looking for a minimum. This time, the fact that a maximum has been found does not imply that a steps has been taken, contrary to the walking algorithm

Once the minimum has been found, we will compute the difference between this value and the previously saved maximum before searching for a new maximum sample. If the difference is greater than a specific threshold, then there will be an update of the step count. Therefore, the step count will be denoted by the transitions from minimum to maximum rather than the number of these points; process done for the step detection during the walking state. In order to exit the running state, the x-axis component of the accelerometer will be continuously checked to see if it is near the resting value for a considerable amount of samples. Below we can see the high level algorithm implemented for the running detection.

```

if (|acc_x-acc_x_rest| < threshold_start_running) --> user has started running
    state_patient = running
    find = maximum                                --> we will be looking for a maximum
    steps = 0

if (state_patient==running)                      --> processing during running

    if(find == minimum)                          --> we are looking for a minimum
        if(previous_sample < present_sample)    --> minimum found
            find=maximum                        -->we will now look for a max
            minimum=previous_sample            --> record the minimum found
            if(minimum-maximum>threshold)-->difference high enough
                steps++

    if(find == maximum)                          --> we are looking for a maximum
        if(previous_sample > present_sample)    --> maximum found
            find=minimum                        -->we will now look for a min value
            maximum=previous_sample            --> record the maximum found

    if (|present_sample- value_at_rest|<threshold_stop_running) --> detect rest
        samples_standing++
        --> if no variation from value at rest for a number of samples
        if(samples_inactivity>certain_number_samples)
            patient_state=standing

```

This will conclude the processing part that involves the detection of the patient when he/she is running.

Falling and Lying Orientation

This will be the last functionality that our application will have. One of the benefits of implementing within our system a fall detection is that, together with the position tracking, we will be able to detect the place where a patient has fallen down and therefore send help immediately. There won't be a need to waste useful time in searching for him/her.

In order to implement this functionality, we first established the following condition in our scenario: a fall occurs when the patient transits directly from a standing to a lying down position. This means that, between these two states, the user will not be sitting down. With

this condition in mind, we then try to think of an approach to be able to detect the four orientations the user is lying down (on his/her back, his/her chest, his/her right arm or left arm).

To do so, we decided to use the readings that come from the SensorTag located in his/her chest, specifically the y and z-axis. Previously we proofed that we were able to detect when the user was lying down on his/her back: we basically determined that if the z-axis component of the accelerometer was greater than a certain threshold then the user transits to the lying down state. Note that if the patient is lying on his/her back the acceleration given by the accelerometer was about 9.8 m/s^2 . Now if we want to determine if the user is lying down on his/her chest, we firstly saw that, in this position, the acceleration value given was around -9.8 m/s^2 . Therefore, to detect when the user is lying on his/her chest, we constantly check the value of the z-axis acceleration component and see if it is approximately equal to that negative acceleration measured. For detecting when the user is lying on his/her left or right arm, we follow a similar process only using this time the y-axis component of the accelerometer.

The above procedure explains then how we can detect the orientation of the user when he/she is lying down. To detect a fall and its consequent orientation, we basically implement the previous algorithm but also checking what the previous state of the patient was. If this state was sitting down, then the user has intentionally lied down however if different, then the user has fallen down. Below we give the high level algorithm for the falling and lying down orientation detection.

```

if (acc_z-acc_z_rest>threshold_lying_front)--> user might be lying down front
    samples_lying++
    position=front

if (acc_z-acc_z_rest<threshold_lying_back)--> user might be lying down back
    samples_lying++
    position=back

if (acc_y-acc_y_rest>threshold_lying_left)--> user might be lying down left
    samples_lying++
    position=left

if (acc_y-acc_y_rest<threshold_lying_right)--> user might be lying down right
    samples_lying++
    position=right

if (samples_lying>certain_value)                --> user has either fell or lied down
    if (patient_state == Sitting)
        fall= NO
    else
        fall=YES

    state_patient = lying

if (state_patient==lying)                        --> processing during lying down

    if (/acc_z-acc_z_rest|<threshold)            --> the user might sitting
        sitting_samples ++

```

```

if(|acc_x_arm-acc_x_arm|<threshold)           --> the user might be standing
    standing_samples ++

if (sitting_samples >certain_value)           --> user is sitting
    state_patient = sitting

if (standing_samples >certain_value)         --> user is standing
    state_patient = standing

```

Notice from the above algorithm how there hasn't been declared a different state of the patient for when he/she falls down. This is because the consequent actions the patient performs after he/she has fallen down are the same as when he/she lies down; he/she will either sit or stand back. That is why we have used the same patient state value for both of this states and used an independent variable denoted as *fall* to determine whether there has been a fall.

This concludes all the activities, along with their algorithms, that our IoT application will be able to detect on a patient with two SensorTags attached to his/her body.

Configure a User Friendly Application on the Server PC

In this part we will give the procedures followed in order to meet the last objective set in our research. The implementation of the actual server database is out of the scope of this report, however we will need to give details on how our gateway sends the data to a PC connected in an outside network.

Every considerable amount of time, our gateway will be sending a message to the server application with the following format. (Note how we use the “@” symbols in order to detect different values for each variable which, ultimately, will be a different column in our database)

```

patient_id@day/month/year@hour:minutes:seconds@temperature
@X_coordinate @Y_coordinate@total_steps@orientation@patient_state

```

In order to send the message, we use a similar procedure to a client and server communication. What we will do is, after a time considered by the user, connect a created socket to the server's database IPv4 address and port number. Once opened, we will build the correct message into an array and send it through the connected socket. After the message has been sent, we close the socket and free the memory assigned to the message array. Basically what we are doing is opening and closing a socket everytime we want to send the message to the server's database application.

We will not show the high level algorithm of this functionality due to its simplicity. However when implementing this part, several challenges arose. First of all, in order to send the message, we had to implement a client (deployed in the Raspberry Pi 2) which used C and a server (deployed in a PC) which used C#. The use of different languages for this scheme brought some compatibility issues in its implementation. Another difficulty that arose was that the Raspberry Pi 2 was not able to acquire visibility to the PC server database application; to test this we used the *ping* command from the Raspberry Pi 2 to the PC server database IPv4 address. One of the alternative solutions to overcome this problem was deactivating the firewall of the PC where the server database was hosted.

Overview & Results

In this section of the report we will show, through screenshots, how our system is able to meet the requirements of a health monitoring application. The structure of this section will go as follows. Firstly, we will show the reader how we have managed to integrate all of the functionalities described in the section above into one single program. In order to do so, we will use a flow graph so the reader has a clear vision of what our health monitoring application is able to do. Once the general overview is shown, the reader will be presented with the different screenshots that provide proof about the completion of our research.

Overview of the Health Monitoring Application

In order to show how our application will integrate all of the functionalities mentioned earlier, we give in the figure below a detailed flow graph along with its description.

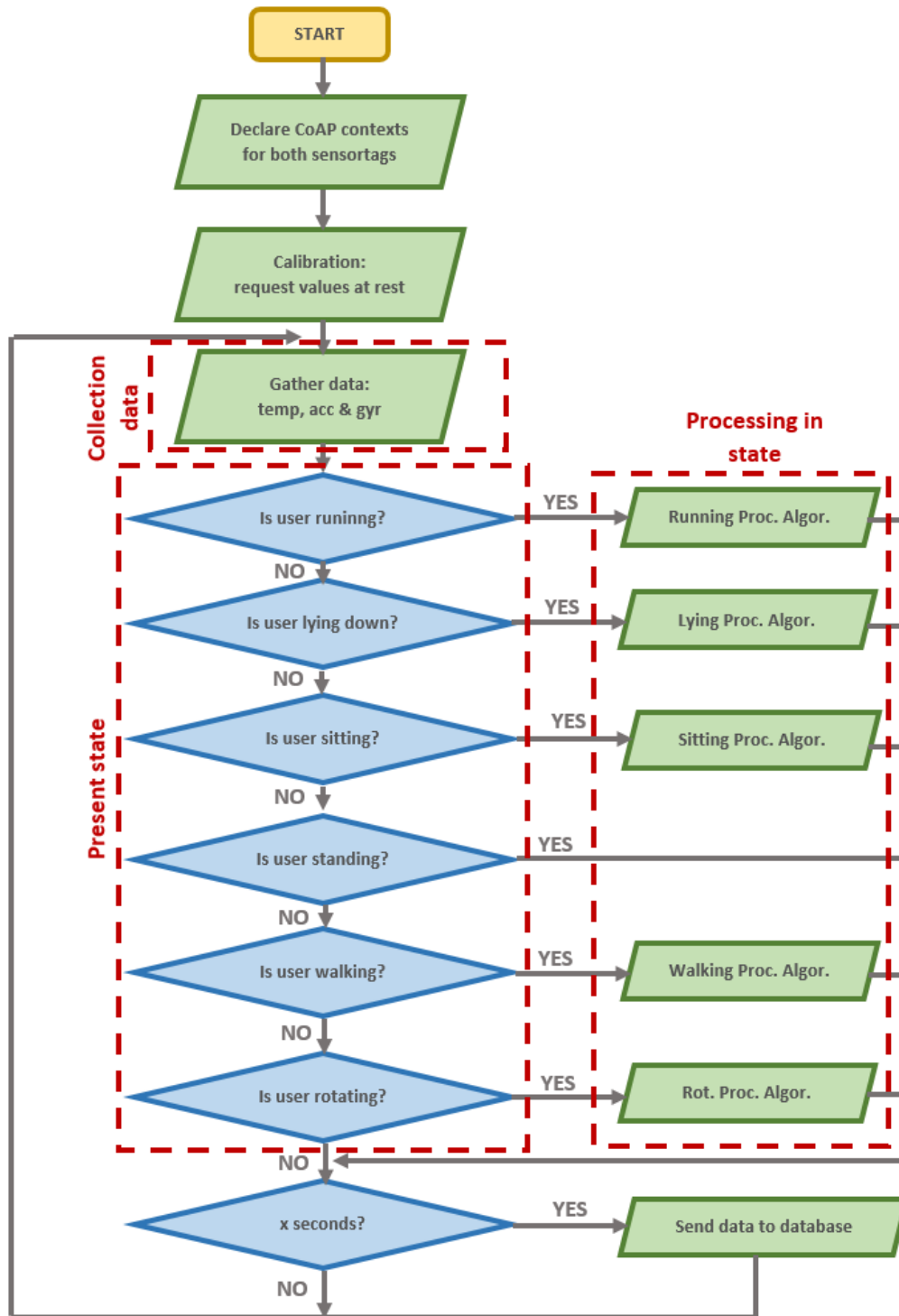


Figure 19: Flowchart of the Health Monitoring Application

From this figure we can see that the program starts off by declaring the CoAP contexts for the two SensorTags and then, it requests the rest values of all sensing elements from the SensorTags. This last statement is used as a calibration process. We register the value of the components at a rest state to take into account the intrinsic error in deciding the patient state. Once these two initial steps are done, we enter into a continuous loop which we can divide into three different sections: collection of data, computation of the present state and the processing within that state.

The collection of values is the first part within this continuous loop. From the values of the components we enter the second phase which is basically a set of conditions that will determine what the present state will be. Once we decide what the patient is actually doing, it is time to perform the processing algorithms, previously described, for each of the states. Notice that when the user is standing, there is no relevant processing to be done and that is why it has been omitted from the flow graph.

After these three sections, the program will check if a number of seconds, decided by the user, have passed from the previous message sent to the server database. If it has, then it will resend a message with the new relevant patient data. Lastly, the program will go back to collect all the data from the SensorTags.

The above is a general overview of how the program will functionally flow within its execution. In the next section we give the actual results of our application.

Results of the Health Monitoring Application

In this part we will simply provide different screenshots that will serve as proof to demonstrate the correct operation of our health monitoring application. The analysis and arguments about these will be left for the next section, *Discussion*.

In figure 20 we see how, through the debugging messages in the Raspberry Pi 2's terminal, the application is able to differ amongst two states: when the user is walking or running. On the left part of the figure, we see how the system has detected that the patient has taken 11 steps and also, how he/she has rotated to an angle of 90°. On the other hand, the image on the right shows that the user has started to run, detecting the 15 steps taken by the patient. The accuracy of both of these processes in detecting correctly steps is 75% and 85% respectively.



Figure 20: Terminal Messages when (left) User Walking and Rotating (right) User Running

From the image above it should be noted that the *out of sync* messages are for debugging purposes and they are printed when one of the CoAP responses has not reached the Raspberry Pi 2.

In figure 21, we provide evidence on how the application, thanks to the implemented processing algorithms, is able to detect when the user is sitting or lying down. Note that when the user sits down, the system interprets, for a small amount of time, that the patient has started walking. This is because for both walking and sitting states, we perform initial similar movements. However, our application is able to detect afterwards if the user is actually sitting down correcting the steps, as printed in the terminal window shown.

```
STARTED WALKING
STEP
Gyr out of sync...
USER SITTING: Fixed errors in steps
Acc out of sync y...
USER LYING DOWN
```

Figure 21: Detection of both Sitting and Lying Down

Concerning processing operations in our Raspberry Pi 2, we show the results regarding the last functionality our gateway is able to perform: lying down and falling down orientation/detection. As we can see on the figure 22 on the left, we have enhanced the lying down detection by providing information about how the patient is lying down. By using this same orientation detection algorithm, we are also able to detect how the patient has fallen down.

```
STARTED WALKING
STEP
USER SITTING: Fixed errors in steps
USER LYING DOWN BACK
USER SITTING
USER LYING DOWN RIGHT
USER SITTING
USER LYING DOWN LEFT
USER SITTING
USER LYING DOWN FRONT
USER SITTING
USER STANDING
```

```
USER RUNNING
Step
Steps corrected: USER FELL DOWN
BACK
USER STANDING
ANGLE 30
STARTED WALKING
USER RUNNING
Step
Steps corrected: USER FELL DOWN
RIGHT
USER STANDING
ANGLE 20
STARTED WALKING
USER RUNNING
Step
Steps corrected: USER FELL DOWN
LEFT
USER SITTING
USER STANDING
ANGLE 20
STARTED WALKING
STEP
USER RUNNING
Step
Steps corrected: USER FELL DOWN
FRONT
```

Figure 22: (left) Lying Down (right) Fall-Down Detection Orientation

It needs to be noted that when the user falls down, for a brief amount of time, the state of the user is thought to be running. This is because, once again, when we fall down and run, the accelerometers in our SensorTags will read similar values. However, after a short time, the application is able to discern these two different states and show the correct condition of the patient.

To conclude, the last objective was for our gateway to be able to communicate with a server database hosted in an IP address. The communication will be based on messages sent from the gateway to the server database which will include all the pertinent information about

the patient. In figure 23 we show how the server database has successfully received and stored the data incoming from the gateway of our WBSN.

The screenshot shows a web application titled "Server App" with a "Data Info" header. The interface includes a sidebar with navigation icons (home, user, location, device) and a main content area titled "Data Information Management".

At the top of the main area, there are input fields for "Record ID: R_Id", "Patient ID: P_Id", "Temperature: Temperature", and "Time Stamp: TimeStamp". Below these are fields for "X: X", "Y: Y", "Steps: Steps", "Orientation: Orienta", and a "Choose Patient's State" dropdown. There are also "Add", "Delete", "Search", and "Refresh" buttons, along with an "AutoRefresh off" toggle.

A hint text reads: "Hint: Mouseover the button to see how to use that fuction." (Note the typo in the original image).

Below the form is a table displaying patient data:

ID	Patient ID	Date+Time	°C	X	Y	Steps	Orientation	State
1	01	18/07/2017-12:25:23	29.84	0.00	0.00	000	0000	WALKING
2	01	18/07/2017-12:25:29	29.84	0.00	5.00	005	0000	RUNNING
3	01	18/07/2017-12:25:33	29.84	0.00	5.00	005	0000	FALL DOWN LEFT
4	01	18/07/2017-12:25:36	29.84	0.00	5.00	005	0000	FALL DOWN LEFT
5	01	18/07/2017-12:25:42	29.84	0.00	5.00	005	0000	SITTING
6	01	18/07/2017-12:25:47	29.84	0.00	5.00	005	0000	ROTATING

Figure 23: Server Database with all the Patient's Information

This closes the *Overview & Results* section of this report. It is now time to discuss these results and other interesting points that arose while developing this investigation.

Discussion

In this part of the report we will discuss several aspects that were brought during the course of the research. The structure of this section will go as follows. Firstly, we will talk about the results gathered along with the some deficiencies we detected in our system. The objective of this first part is to open a path for improvement as a future work. After this, we will bring out ideas on possible enhancements for our health monitoring application which could be taken, once again, as a future work.

Discussion of Results

From the results shown in the previous section, it can be easily seen that our application meets, at least, with the minimum requirements needed for health monitoring. It is able to detect all of the states proposed in the objectives and gives fairly good accurate values in doing so. However, when debugging the system, some inefficiencies were found.

One of the most notorious ones was the synchronization problems in our application. When running tests it was seen that losing packets in the communication between the gateway and the sensor nodes was fairly common, especially when we were positioned far away from the gateway. We strongly believe that the main problem behind this issue resides in the quality of the sensor nodes used.

The CC2650 SensorTags are pretty cheap devices. They tend to break easily (throughout this research three SensorTags had several problems) and the antenna they are built with is of poor quality. In fact, when the antenna of the SensorTag does not face the OpenLabs module of the gateway, the packet loss tends to be much higher than when it is. Thus, the radiofrequency directionality of this device is not quite good and therefore, if we want our application to work better, we will need to substitute this sensing node.

A possible alternative could be a Libelium Wasp mote sensor integrated with the IBM Mote Runner SDK [22], shown in the figure on the right. These sensor nodes, both have high quality radio devices (the AT86RF231), many of the needed sensors as well as a number of pins that will allow for additional sensors to be inserted. Now, it has to be said that the problem with the transmission may not reside in the SensorTags but may be on the gateway. If this was the problem, we could substitute the OpenLabs module for the AT86RF231 transceiver in our Raspberry Pi 2.



Figure 24: Libelium Wasp mote + IBM Mote Runner

Another improvement that could be made within our system concerns the walking detection algorithm. As we know, in order to detect the number of steps, we perform a thresholding technique. We basically see if the accelerometer data is above or below a certain value and, if it is, we determine a step is taken. This alternative works very well for estimation purposes however, if we wanted to improve accuracy we will need to deploy a stronger processing algorithm.

One of the ideas thought, and which could be used as a future work for this research, is machine learning. We could use, for instance, an artificial neural network to model the

behavior during our walking state and thus determine whether there has been a step or not. This will imply having to train the network with a number of sets from which it could learn from and decide these two possible outputs. Also it has to be said that the machine learning algorithm implementation doesn't need to be restricted to the walking state. We could also apply this approach to the whole application, having the neural network decide, from the data requested from the sensors, what is the condition of the patient (walking, running, standing...).

It has to be said that the approach mentioned above, while it may give higher accuracy, it will also imply a much heavier processing. Therefore when implementing this solution, we will have to see if the increase in accuracy is worth the processing overhead in our Raspberry Pi 2. For instance, while in detecting steps during walking the range of improvement is still desirable during running we might just content ourselves with the results achieved.

These are the possible modifications we could make in our health monitoring application in order to improve both its accuracy and efficiency.

Discussion of Enhancements

Apart from the improvements we can make on the existing functions of our health monitoring application, we can also implement other ideas within. In this sub-section we will discuss possible enhancements that could be left as a future work for our application and which will make it more valuable and robust for our users.

Sleep tracking [23] is becoming a great concern nowadays. A report recently divulged by the CDC states that sleep related problems are now becoming a public health epidemic. The indirect causes of not gaining a good night sleep involve, amongst many, fatal car accidents, depressions, possible sleep apnea or a decrease in the economical activity. Therefore we believe that implementing some sort of functionality within our application that could track a patient's rest is of great importance.

A starting point for this could be monitoring the breathing rate of a patient who is sleeping. From this data extracted we could be able to discern between the different stages that make up one sleeping cycle. This cycle, which has a total duration of 90 minutes, is composed of five stages, the first four being non REM and the last stage REM. Non-REM stages are defined by a regular breathing rate in both amplitude and frequency, having a lower regular frequency in the last two stages. On the other hand, the REM stage is defined by irregular breathing in both of the previous magnitudes.

The aim of this new implemented function will reside basically in making sure that the patient transits through all of these stages as well as determine the time spent in each. This way, we can monitor and determine whether the user has had a night from which he/she is full recovered. Now, in order to measure this breathing rate we propose the following approach: adhere a SensorTag to the abdominals of the patient and extract its accelerometer data. The abdominals, during breathing, are subjected to a high movement which is able to be perceived by our device and thus extract the appropriate information. In figure 25 below we give a small plot in which the accelerometer data from a SensorTag adhered to the abdominals of the user has been extracted. Notice how the peaks and valley both indicate when the user is breathing in and out respectively.

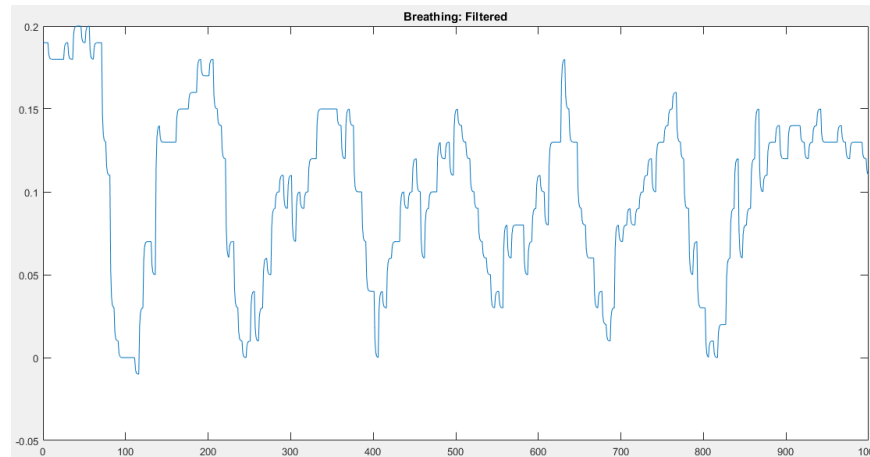


Figure 25: Accelerometer Data of Patient during Breathing

Although it is true that the accuracy of this SensorTag is not good enough and the breathing performed are exaggerated, these results opens a path to implementing the desired function. If we were to find a SensorTag with a higher precision than the one present, we could be able to plot this data in better conditions and, by examining its frequency and amplitude, track the sleeping activity of a patient.

Another aspect which it will be important to work in the future is all regarding with the data transmission between the gateway and the server database application. At the moment, the communication between these two platforms are made wire-to-wire meaning that most of the problems such as data loss, corruption or security have been overlooked. However, we must not forget that these problems due exist, especially when working with wireless connectivity, and therefore the consequent actions have to be taken in order to make our health monitoring application more robust.

While packet loss during the transmission between the gateway and the server database application is something inevitable, there are some contingency procedures that will allow us to reduce the damage done if this event occurs. A simple mechanism will be to implement in the Raspberry Pi 2 a limited size buffer. This buffer will store the messages that have been sent to the server database application and wait for a confirmation message from the server database. If the confirmation message doesn't reach the gateway within a time interval, the packet will be interpreted as lost and a copy from the buffer will be resent. On the other hand if the confirmation is received, the packet will be erased from the buffer giving up more available space.

The above approach will be efficient against packet loss however it will not be able to cope with packet corruption. Packet corruption is understood as the unintentional modification of a packet during its transmission in the medium. Nevertheless, an intentional modification coming from a network attacker is also possible and thus we should consider as well the security of our health monitoring application. Also our application transmits private information about a patient, something which should be taken into consideration when improving the security of our system. Thus, we thought of implementing a security algorithm that will offer users the following characteristics:

- **Confidentiality:** the information transmitted between the gateway and the server database is only visible to these two and not any other patients within the network.
- **Integrity:** the information transmitted between the gateway and the server database is secure from not authorized modifications.
- **Authentication:** the source that declares to have transmitted the information is the actual source.

In order to maintain the integrity of the package through the communication, we will implement a message digest (or hash) in the Raspberry Pi 2, for instance MD5. Message digest functions have perfect characteristics to assure the integrity of a message. The first one is that they are a one-way function meaning that it is impossible from the hash of the message to compute the actual message. The second characteristic is that it is unfeasible to find two messages that will give the same hash.

Therefore, in order to meet the integrity in our communication, we will transmit the following message everytime we send the patient's information from the Raspberry Pi 2 to the server database application. (*Note that $H(m)$ means that the hash algorithm has been applied over the clear message*)



Figure 26: Syntax of Message to Meet Integrity

Once the server database receives this message, it will compute $H(m)$ from the m received and see if it is equal to $H(m)$. If it is, then we can be sure that our message has not changed its content throughout its communication. Now, imagine there was an attacker or an interference in the medium that caused m to be modified; we will denote the modified version as m' . Therefore, once $[m', H(m)]$ reaches the server database it will calculate $H(m')$ from m' and notice that it is not equal to $H(m)$ received, invalidating the message for its acceptance. From this scenario we can then see that, by appending the hash of the message, we can guarantee its integrity.

Lastly, in order to also support authenticity and confidentiality in the communication between the gateway and the server database, we will modify the previous message to the following.

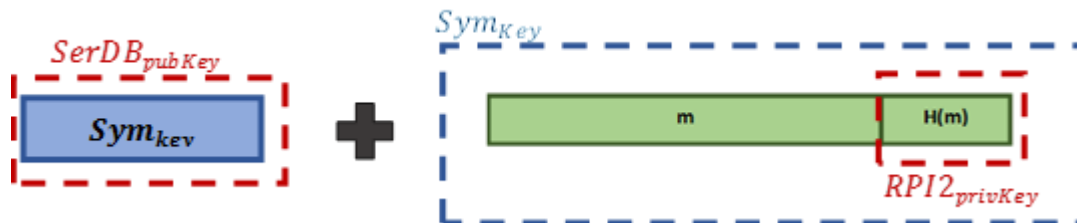


Figure 27: Hybrid Encryption Scheme

The above figure shows that we first need to generate a symmetric key that will encrypt the $[m, H(m)]$. Once this process is done, the symmetric key will then be encrypted with the server database's public key. This way, we guarantee that only the server database is the one

able to acquire the value of Sym_{key} needed to decrypt the message. Therefore, confidentiality is achieved: the contents are not sent clear throughout the transmission and are only visible to the end point of the communication.

Now, notice how we are sending the hash of the message but encrypted with the Raspberry Pi 2's private key. This acts a signature for the gateway: the Raspberry Pi 2 is the only one that possess its private key. Thus, when the server database receives this message it will first compute $H(m)$ and then decrypt, with the gateway's public key, the appended part received. It will compare the two values and if they are equal, the end point can safely say that the message comes in fact from the Raspberry Pi 2.

The above implementation, due to all of the encryption and decryption, will of course bring some computation overhead in our processing node. Also, we will have to establish procedures to securely transmit the Raspberry Pi 2's and the server database's private key; they should be the only ones who know its value in the network. Despite these small disadvantages, the benefits that this scheme will bring, from a security point of view, will be of great value to our application.

This completes the discussion of new functionalities that we could implement in our system.

Conclusion

With this last section, we conclude the research of *Implementation of 6LoWPAN (IPv6) for Wireless Body Sensor Network (WBSN)*.

The architecture implemented to deploy our system is based on a gateway, a Raspberry Pi 2, which requests data from the sensors that integrate the WBSN. In order to receive this data there has been two approaches shown. One of them uses the Raspberry Pi 2 as a border router, by deploying the 6lbr application, and has a SensorTag attached to its USB port which operates as a transceiving antenna. On the other hand, we have successfully implemented an own solution using an OpenLabs module attached to the GPIO pins of the Raspberry Pi 2. This last alternative is more efficient (not necessary for 6lbr to be deployed) and is lower in cost. Therefore, this option has been chosen in implementing our WBSN application.

Once the data reaches the gateway it will be subjected to processing. Through this processing, we are able to detect different physical conditions on the patient e.g. walking, running, falling down, sitting down...By monitoring these states, we are able to study the behavior of the patient as well as track him/her in case we need to send in medical staff if he/she falls down. This processed information will be made available over the Internet by being sent to a server database from which the health institution will have access to.

After implementing these processing algorithms, several tests were made in order to verify their correct operation. Despite the successfulness in the results, there is always room for improvement and which can be a starting point for future investigations. For instance, we could improve the accuracy in our application by implementing more complex algorithms. One of the alternatives could be machine learning in which we will train this algorithm with sets of data for it to later make accurate predictions on the state of the patient. However, not only can we improve the functions of our application but also enhance the scope of them. An interesting idea to implement will be sleep monitoring, in which a sensor is attached to the abdominals of the patient to determine, from his/her breathing rate, if he/she has had a good rest during the night.

The different functionalities we can implement in a WBSN are numerous and from which health areas can greatly benefit from. Not only they reduce healthcare costs in health institutions but they also make the work of medical staff much more efficient. After this research, we strongly believe that WBSN will be very present in the health environment since they will constitute an indispensable tool.

References

- [1] "A simple Explanation of 'The Internet of Things'". (2014). *Forbes*. [Online] Available at: <https://www.forbes.com/sites/jacobmorgan/2014/05/13/simple-explanation-internet-things-that-anyone-can-understand/#79f0c2741d09> [Accessed 13 Feb. 2017].
- [2] T. Gonnot, W. J. Yi, E. Monsef and J. Saniie, "Robust framework for 6LoWPAN-based body sensor network interfacing with smartphone," *2015 IEEE International Conference on Electro/Information Technology (EIT)*, Dekalb, IL, 2015, pp. 320-323.
- [3] W. J. Yi, O. Sarkar, T. Gonnot, E. Monsef and J. Saniie, "6LoWPAN-enabled fall detection and health monitoring system with Android smartphone," *2016 IEEE International Conference on Electro Information Technology (EIT)*, Grand Forks, ND, 2016, pp. 0174-0178.
- [4] T. Gonnot, W. J. Yi, E. Monsef, P. Govindan and J. Saniie, "Sensor network for extended health monitoring of hospital patients," *2014 IEEE Healthcare Innovation Conference (HIC)*, Seattle, WA, 2014, pp. 236-238.
- [5] Adafruit, "Raspberry Pi Model 2 B Datasheet" (2015). [Online] Available at: <http://www.adafruit.com/pdfs/raspberrypi2modelb.pdf> [Accessed 25 July. 2017].
- [6] Texas Instruments, "Multi-Standards CC2650 SensorTag Design Guide" (2015). [Online] Available at: <http://www.ti.com/lit/ug/tidu862/tidu862.pdf> [Accessed 25 Jul. 2017].
- [7] Texas Instruments, "CC2650 SimpleLink™ Multistandar Wireless MCU." (2015). [Online] Available at: <http://www.ti.com/lit/ds/symlink/cc2650.pdf> [Accessed 25 Jul. 2017].
- [8] OpenLabs, "Raspberry Pi 802.15.4 radio." 2014. [Online] Available at: <http://openlabs.co/store/Raspberry-Pi-802.15.4-radio> [Accessed 1 Mar. 2017].
- [9] J. Schöwälder "Internet of Things: 802.15.4, 6LoWPAN, RPL, COAP" (2010) [Online] Available at: <https://www.utwente.nl/ewi/dacs/colloquium/archive/2010/slides/2010-utwente-6lowpan-rpl-coap.pdf> [Accessed 15 May. 2017].
- [10] D. E. Culler, J. Hui, "6LowPAN Tutorial: IP on IEEE 802.15.4 Low-Power Wireless Networks." [Online] Available at: <https://pdfs.semanticscholar.org/b46e/6f9a777e306f2e7e333e1a41383f0e41636a.pdf> [Accessed 15 May. 2017].
- [11] J. Lukkien, R. Verhoeven (2016), "Internet of Things 2015/2016" [Online] Available at: <http://www.win.tue.nl/~johanl/educ/2IMN15/IoT-13-6LoWPAN.pdf> [Accessed 15 May. 2017].
- [12] CoAP Technology, "RFC 752 Constrained Application Protocol" [Online] Available at: <http://coap.technology/> [Accessed 20 May. 2017].
- [13] Github-Contiki, "Contiki Repository" [Online] Available at: <https://github.com/contiki-os/contiki> [Accessed 10 Mar. 2017].
- [14] M. Caraccio, "Getting Started with TI 6LowPAN SensorTag" (January 4th, 2016) [Online] Available at: <http://piratefatche.ch/getting-started-with-ti-6lowpan-sensortag/>

[Accessed 10 Mar. 2017].

[15] A. Liñán, Vives, M. Zennaro, A. Bagula, and E. Pietrosemoli (2016), “Internet of Things (IoT) in 5 days.” [Online] Available at: http://wireless.ictp.it/school_2016/book/IoT_in_five_days-v1.0.pdf [Accessed 15 Mar. 2017].

[16] Github-6lbr, “6lbr Repository” [Online] Available at: <https://github.com/cetic/6lbr/wiki> [Accessed 10 Apr. 2017].

[17] GitHub-6LoWPAN, “How to install 6LoWPAN Linux Kernel on Raspberry Pi” (2017) [Online] Available at: <https://github.com/RIOT-OS/RIOT/wiki/How-to-install-6LoWPAN-Linux-Kernel-on-Raspberry-Pi> [Accessed 10 Mar. 2017].

[18] Raspbian Lite Version [Online] Available at: https://downloads.raspberrypi.org/raspbian_lite/images/raspbian_lite-2017-02-27/2017-02-16-raspbian-jessie-lite.zip [Accessed 20 Feb. 2017].

[19] Libcoap, “libcoap: C-Implementation of CoAP” [Online] Available at: <https://libcoap.net/> [Accessed 20 May. 2017].

[20] R. Zhi “A Drift Eliminated Attitude & Position Estimation Algorithm in 3D” (2016) *Graduate College Dissertations and Theses*. Paper 450.

[21] K. Seifert and O. Camacho, “Implementing Positioning Algorithms Using Accelerometers” *NXP*. [Online] Available at: <http://www.nxp.com/docs/en/application-note/AN3397.pdf> [Accessed 18 Jun. 2017].

[22] Liblium, “Waspnote Mote Runner 6LoWPAN Development Platform” [Online] Available at: <http://www.libelium.com/products/waspnote-mote-runner-6lowpan/> [Accessed 25 Jul. 2017].

[23] J. Mann “The Ultimate Guide to Sleep Tracking” (2017) [Online] Available at: https://sleepjunkies.com/features/the-ultimate-guide-to-sleep-tracking/#Why_track_sleep [Accessed 25 Jul. 2017].