SENSOR INTEGRATION FOR SMART CITIES USING MULTI-HOP NETWORKS

Bruno de Sá Moreira Fernandes

Electrical Engineering Department

Master in Electronic and Computer Engineering

Specialization in Telecommunications

2015

This report fulfills the needed requirements of the course Tese/Dissertação (Thesis) from the 2nd year of the Master in Electronics and Computer Engineering.

Student: Bruno de Sá Moreira Fernandes, Nr. 1080557, 1080557@isep.ipp.pt
Scientific Orientation: Prof. Dr. Jorge Botelho da Costa Mamede, jbm@isep.ipp.pt

Company: Telecommunications Institute

Supervision: Prof. Dr. Tânia Cláudia dos Santos Pinto Calçada, tcalcada@fe.up.pt

Electrical Engineering Department
Master in Electronic and Computer Engineering
Specialization in Telecommunications

2015

To my parents.

Acknowledgements

Throughout this dissertation work there was the contribution of people without which this work would not have the quality here presented.

I leave here a big appreciation to my supervisor, Prof. Dr. Tânia Calçada, for all her contribution, effort, dedication, shared knowledge and her high quality standards of work. I also appreciate the vote of confidence for accepting me and to provide me a quickly integration in such a huge project that Future Cities is.

Another huge appreciation goes to my thesis advisor Prof. Dr. Jorge Mamede, for the vote of confidence when readily agreed to be my advisor. For all the tips he gave me, for his high quality standards of work and for all the guidance and support either academic or moral throughout the development of this dissertation work.

A big appreciation to both for showing me that knowledge knows no boundaries and for turning my future life goals way more ambitious by awakening in me the interest to pursue a research career.

I leave a thank you to all PhD students from Telecommunications Insitute by sharing with me their ideas and opinions regarding this work.

I leave a thank you to my parents and close friends by their time, readiness and help they provided me in some of the field tests performed throughout this work.

I leave my huge appreciation to all collaborators from the Future Cities Project and Telecommunications Institute for accepting me and look at me as a contributer. This feeling kept me motivated most of the time. I also appreciate the exceptional environment created in the Shannon Lab.

My last big aprecition goes to Prof. Dr. Vladimiro Machado for his motivation in the early beginnings and the kickstart that allowed me to get this far.

Bruno Fernandes

Abstract

Smart Cities are designed to be living systems and turn urban dwellers life more comfortable and interactive by keeping them aware of what surrounds them, while leaving a greener footprint. The Future Cities Project [1] aims to create infrastructures for research in smart cities including a vehicular network, the *BusNet*, and an environmental sensor platform, the Urban Sense. Vehicles within the *BusNet* are equipped with On Board Units (OBUs) that offer free Wi-Fi to passengers and devices near the street. The Urban Sense platform is composed by a set of Data Collection Units (DCUs) that include a set of sensors measuring environmental parameters such as air pollution, meteorology and noise. The Urban Sense platform is expanding and receptive to add new sensors to the platform. The parnership with companies like *TNL* were made and the need to monitor garbage street containers emerged as air pollution prevention. If refuse collection companies know prior to the refuse collection which route is the best to collect the maximum amount of garbage with the shortest path, they can reduce costs and pollution levels are lower, leaving behind a greener footprint.

This dissertation work arises in the need to monitor the garbage street containers and integrate these sensors into an Urban Sense DCU. Due to the remote locations of the garbage street containers, a network extension to the vehicular network had to be created. This dissertation work also focus on the Multi-hop network designed to extend the vehicular network coverage area to the remote garbage street containers. In locations where garbage street containers have access to the vehicular network, Roadside Units (RSUs) or Access Points (APs), the Multi-hop network serves has a redundant path to send the data collected from DCUs to the Urban Sense cloud database. To plan this highly dynamic network, the Wi-Fi Planner Tool was developed. This tool allowed taking measurements on the field that led to an optimized location of the Multi-hop network nodes with the use of radio propagation models. This tool also allowed rendering a temperature-map style overlay for Google Earth [2] application. For the DCU for garbage street containers the parner company provided the access to a *HUB* (device that communicates with the sensor inside the garbage containers). The Future Cities use the Raspberry pi as a platform for the DCUs. To collect the data from the *HUB* a RS485 to

RS232 converter was used at the physical level and the *Modbus* protocol at the application level. To determine the location and status of the vehicles whinin the vehicular network a *TCP Server* was developed. This application was developed for the OBUs providing the vehicle Global Positioning System (GPS) location as well as information of when the vehicle is stopped, moving, on idle or even its slope. To implement the Multi-hop network on the field some scripts were developed such as *pingLED* and "shark". These scripts helped upon node deployment on the field as well as to perform all the tests on the network. Two setups were implemented on the field, an urban setup was implemented for a Multi-hop network coverage survey and a sub-urban setup was implemented to test the Multi-hop network routing protocols, Optimized Link State Routing Protocol (OLSR) and *Babel*.

Keywords

Multi-hop, Ad-hoc, Network, Protocols, MIPS, ARM, Wi-Fi, Radio Propagation Models.

Resumo

As *Smart Cities* (cidades inteligentes) são cidades projectadas para serem sistemas vivos, com a capacidade de transformar a vida dos seus habitantes, proporcionando-lhes um maior conforto e interatividade mantendo-os conscientes do que os rodeia, deixando em simultâneo uma pegada mais verde. O Projeto Future Cities [1] visa a criação de infra-estruturas para a investigação em cidades inteligentes, incluindo uma rede veicular, a *BusNet*, e uma plataforma de sensores ambientais, a *Urban Sense*. Os veículos da *BusNet* estão equipados com unidades a bordo (OBUs) que oferecem acesso Wi-Fi gratuito aos passageiros e dispositivos próximos da rua. A plataforma *Urban Sense* é composta por um conjunto de unidades colectoras de dados (DCUs) que incluem um conjunto de sensores que medem parâmetros ambientais, tais como: poluição do ar, meteorologia e ruído. A plataforma *Urban Sense* está em expansão e receptiva para adicionar novos sensores à plataforma. Parcerias com empresas como a *TNL* foram feitas com a necessidade de monitorizar os contentores de lixo na rua. Esta iniciativa surgiu como prevenção da poluição do ar. Se as empresas de recolha de lixo souberem antecipadamente à recolha, qual é a melhor rota para recolher o máximo de lixo utilizando o trajecto mais curto, esta informação pode reduzir os custos e baixar os níveis de poluição, deixando para trás uma pegada mais verde.

Esta dissertação surge com a necessidade de monitorizar os contentores de lixo da rua e integrar estes sensores numa DCU para a plataforma *Urban Sense*. Devido à localização remota dos contentores de lixo da rua, teve de ser proposta uma extensão para a rede veicular já existente. A extensão proposta é uma rede *Multi-hop*, projetada para aumentar a área de cobertura da rede veicular para abranger os contentores de lixo remotos. Em locais onde os contentores de lixo tenham acesso à rede veicular, unidades próximas da estrada (RSUs) ou pontos de acesso públicos (APs), a rede *Multi-hop* serve como um caminho redundante para enviar os dados recolhidos a partir das DCUs até à base de dados na *cloud* da *Urban Sense*. Para planear esta rede altamente dinâmica, a ferramenta *Wi-Fi Planner Tool* foi desenvolvida. Esta ferramenta permitiu efectuar medições da potência de sinal no terreno que levaram a uma localização optimizada dos nós da rede *Multi-hop* com a utilização dos modelos de propagação de rádio. Esta ferramenta também permitiu gerar um

mapa térmico de sobreposição para a aplicação Google Earth [2]. Para a DCU dos contentores de lixo, a empresa parceira disponibilizou o acesso a um *HUB* (dispositivo que comunica com o sensor no interior dos contentores de lixo). A *Future Cities* usa o *Raspberry pi* como plataforma para as DCUs. Para recolher os dados do *HUB* foi utilizado um conversor RS485 para RS232 ao nível físico e o protocolo *Modbus* ao nível de aplicação. Para determinar a localização e o estado dos veículos dentro da rede veicular foi desenvolvida uma aplicação *TCP Server*. Esta aplicação foi desenvolvida para as OBUs fornecendo o Sistema de Posicionamento Global (GPS) do veículo, bem como informações de quando o veículo está desligado, em movimento, parado ou até mesmo a sua inclinação. Para implementar a rede *Multi-hop* no terreno foram desenvolvidos alguns *scripts*, como o *pingLED* e o "shark". Estes *scripts* ajudaram na localização dos nós no terreno, bem como para realizar todos os testes na rede. Duas configurações foram implementadas no terreno, uma configuração urbana foi implementada para um levantamento da cobertura da rede *Multi-hop* e uma configuração sub-urbana foi implementada para testar os protocolos de encaminhamento da rede *Multi-hop*, "Optimized Link State Routing Protocol (OLSR)" e *Babel*.

Palavras-Chave

Multi-hop, Ad-hoc, Rede, Protocolos, MIPS, ARM, Wi-Fi, Modelos de Propagação de rádio.

Résumé

Les *Smart Cities* sont conçues comme des systèmes urbains procurant un mode de vie plus confortable et interactif aux habitants dans le sens où elles permettent de renseigner ces derniers sur leur environnement et ce sans avoir de conséquences néfastes sur celui-ci. *Future Cities* Project [1] vise à créer des infrastructures de recherche dans ces *Smart Cities*. Ces infrastructures prévoient la mise en place d'un réseau routier appelé le *BusNet,* et d'une plateforme accueillant des capteurs environnementaux, L'*Urban Sense*. Les véhicules au sein du réseau *BusNet* sont équipés à bord d'unités (On Board Units - OBUs) offrant aux passagers la possibilité de se connecter gratuitement à la Wi-Fi et à d'autres appareils implantés dans le paysage urbain. La plateforme *Urban Sense* est composée par un ensemble de capteurs récoltant une multitudes des données environnementales tels que la pollution de l'air, des données météorologiques et les nuisances sonores. Cette plateforme connait un développement considérable et l'ajout de nouveaux capteurs est envisageable. Des partenariats avec des entreprises comme *TNL* ont vu le jour et la nécessité d'assurer un meilleur suivi de ces conteneurs à ordure se fait autant ressentir que le besoin de prévention contre la pollution dans l'air. Une meilleure connaissance en amont de la part des entreprises concernant la collecte d'ordures, et des itinéraires favorisant la plus grande collecte de déchets par le chemin le plus court favoriserait une réductions des coûts et du niveau de pollution, et permettrait donc aux entreprises d'exercer une emprunte plus écologique.

Cette thèse a pour but de promouvoir un meilleur suivi des conteneurs à ordures et de proposer l'intégration de ces capteurs dans l'*Urban Sense* DCU. En raison de l'éloignement de certains conteneurs, une extension du réseau au réseau automobile a du être pensée. Ce travail de recherche se penche également sur le réseau Multi-hop conçu pour étendre la zone de couverture des conteneurs à ordures éloignés du réseau routier. Pour les endroits où les conteneurs à ordures sont situés sur le réseau routier, au niveau des unités situés en bord de route ou sur les points d'accès, le réseau Multi-hop assure la fonction d'envoyer les données collectées par les capteurs de l'*Ubran Sense* vers le cloud de l'*Urban Sense*. Un outil de planification par Wi-Fi a été développé afin d'assurer la gestion de ce réseau ultra dynamique. Cet outil a permis de collecter sur le terrain, à travers

la propagation des ondes, des mesures qui ont conduit à l'optimisation des nœuds du réseau Multi-hop. Il permet également la traduction des données en une carte thermique transposable sur Google Earth [2]. Les cités du future (*Future Cities*) utilise pour les DCUs une plateforme appelé *Raspberry Pi*. Afin de collecter les données d'un *HUB*, un convertisseur RS485 pour RS232 a été utilisé pour les questions d'ordre physiques, et un protocole *Modbus* pour son application. Une application *TCP Server* a été mis en œuvre pour déterminer l'emplacement et le statu des véhicules au sein du réseau. Cette application a été développé pour les OBUs qui renseigne de la position géographique du véhicule et fournissent des informations sur l'état du véhicule: arrêté, en mouvement, au ralenti, l'inclinaison de celui-ci. Des scripts tels que *pingLED* et "shark" on été développé pour mettre en œuvre le réseau Multi-hop sur le terrain. Ces scripts nous ont facilité la tâche lors du déploiement du nœud mais aussi pour effectuer l'ensemble des tests sur le réseau. Deux types de configurations ont été prévu dans la mise en œuvre, une configuration urbaine pour pallier à la couverture de réseau Multi-hop, et une configuration périphérique afin de tester les protocole d'acheminement du réseau Multi-hop, "Optimized Link State Routing Protocol (OLSR)" et *Babel*.

Mots-clés

Multi-hop, Ad-hoc, Réseau, Protocoles, MIPS, ARM, Wi-Fi, Modèles de propagation radio.

Kurzfassung

Die *Smart Cities* sind konzipiert um lebende Systeme darzustellen, und definiert das Leben von Stadtbewohnern bewusst konfortabler und interaktiver zu gestalten, ohne das Kompromiss eines Lebensfreundlichen Grünen Punkt zu gefährden. Der *Future Cities* Projekt [1] Zielt auf die erzeugung von Infrastrukturen für die Forschung in *Smart Cities*, einschließlich eines Fahrzeugnetzwerk, das *BusNet* und Umweltsensorplattform, die *Urban Sense*. *BusNet* Fahrzeuge sind mit On Board Units (OBUs) ausgestattet das den Fahrgästen und allen Geräten in der Umgebung ein kostenfreien Wi-Fi bieten. Das *Urban Sense* Palttform ist aus einer Reihe von Data Collection Units (DCU) komponiert die für sich eine reihe von Sensoren beeinhalten die Umweltparameter wie: Luftverschmutzung, Meteorologie und Lärm. Das *Urban Sense* Palttform ist in Erweiterung und aufnahmefähig um neue Sensoren zur Plattform hinzufügen. Das parnership mit Unternehmen wie *TNL* hat sich mit der Notwendigkeit von einer Müll Container überwachung erstellt. Diese Initiative erschien mit der Vorbeugung der Luftverschmutzung. Wenn die Müllabfuhrunternehmen schon im Voraus wissen welche ist die bessere und kürzerste strecke um die maximale Menge an Müll zu sammeln, dan kann es zu einer Kosten und Unweltbelastung geringerung fügen.

Diese Dissertation stellt sich in der Notwendigkeit, die Müllcontainer zu überwachen und diese Sensoren in das *Urban Sense* Palttform DCU zu integrieren. Wegen der entfernten Stellen der Müllcontainer, musste eine Netzwerkerweiterung an dem Fahrzeugnetzwerk erschaffen werden. Diese Dissertationsarbeit fokussiert sich auch auf das *Multi-hop* Netzwerk, das für das erweitern der Fahrzeug-Netzabdeckung der Müll Fahrzeuge entwickelt wurde. In Zonen wo die Müllcontainer zugriff auf das Fahrzeugnetzwerk haben, Strassenrand-Units (RSUs) oder Access Points (APs), dient das *Multi-hop* Netzwerk als redundanten Pfad um die Daten von der DCUs aus bis zur *Urban Sense* Cloud zu senden. Um dieses hochdynamische Netz zu planen wurde das *Wi-Fi Planner Tool* entwickelt. Dieses Werkzeug erlaubt Messungen auf dem Feld, die zu einer optimierten Anordnung der *Multi-hop* Netzwerkknoten unter Verwendung von Funkausbreitungsmodelle geführt. Dieses Tool erlaubt auch das generieren von einer temperatur Karte durch überlappung für die App Google Earth [2]. Für die DCU der Müllcontainer hat der Geschäfftspartner ein

zugriff auf ein *HUB* (Gerät, das mit dem Sensor in den Müllcontainer kommuniziert) bereit gestellt.Die *Future Cities* nutzen die *Raspberry Pi* als Plattform für die DCUs. Um die Daten aus der *HUB* zu entnehmen wurde ein Konvertorvon RS485 zu RS232 auf der physikalischen Ebene verwendet und das *Modbus*-Protokoll auf der Anwendungsebene. Um den Ort und Zustand der Fahrzeuge innerhalb des Fahrzeugnetzwerk zu ermitteln wurde eine *TCP Server* App entwickelt. Diese Anwendung wurde für die OBUs entwickelt, und gibt Informationen aus wie: Global Positioning System (GPS), ob das Fahrzeug angehalten wurde, ob es sich bewegen, oder sogar seine Steigung. Um das *Multi-Hop*-Netzwerk in der Praxis zu implementieren wurden einige Scripts entwikelt: *pingLED* und “shark”. Diese Scripts halfen beim lokalisieren von den Knoten auf den Fächen, wie auch um alle Tests im Netzwerk durchzuführen. Zwei Ansätze wurden auf dem Gebiet durchgeführt, eine städtische um eine *Multi-Hop*-Netzabdeckung zu untersuchen und ein Sub-städtische um die *Multi-Hop*-Netzwerk-Routing-Protokolle zu überprüfen, Optimized Link State Routing Protocol (OLSR) und *Babel*.

Schlüsselwörter

Multi-hop, Ad-hoc, Netzwerk, Protokolle, MIPS, ARM, Wi-Fi, Funkausbreitungsmodelle.

Contents

List of Figures

List of Tables

List of Code excerpts

Abbreviations

ADU - Application Data Unit

AODV - Ad-hoc On-Demand Distance Vector

AP - Access Point

ARM - Advanced RISC Machines

ARP - Address Resolution Protocol

BATMAN - Better Approach to Mobile Ad-hoc Networking

BSS - Basic Service Set

BSSID - Basic Service Set Identification

CA - Certificate Authority

CRC - Cyclic Redundancy Check

CSV - Comma Separated Value

CTS - Clear to Send

DB - Database

DCU - Data Collection Unit

DHCP - Dynamic Host Configuration Protocol

DNS - Domain Name Service

DSDV - Destination-Sequenced Distance Vector

DSR - Dynamic Source Routing

DSRC - Dedicated Short-Range Communications

DTN - Delay and Disruption-Tolerant network

ECG – Electrocardiogram

EIGRP - Enhanced Interior Gateway Routing Protocol

EIRP - Effective Isotropic Radiated Power

EU - European Union

FCC - Federal Communications Commission

FIRE - Future Internet Research and Experimentation

FP7 - Seventh Framework Programme

FSR - Fisheye State Routing

GPIO - General Purpose Input/Output

GPRS - General Packet Radio Service

GPS - Global Positioning System

GPU - Graphics Processing Unit

GSM - Global System for Mobile Communications

HMI - Human Machine Interface

HNA - Host and Network Association

HSLS - Hazy Sighted Link State

HWMP - Hybrid Wireless Mesh Protocol

I/O - Input/Output

I^2C - Inter-Integrated Circuit

IBSS - Independent Basic Service Set

IC - Integrated Circuit

ICMP - Internet Control Message Protocol

ICT - Information and Communication Technologies

ID - Identity

IHU - I Heard You

IoT - Internet of Things

IP - Internet Protocol

IPSec - Internet Protocol Security

ISM - Industrial Scientific Medical

LED - Light-Emitting Diode

LMSC - LAN/MAN Standards Committee

LoRA - Long Range

LOS - Line Of Sight

LRC - Longitudinal Redundancy Check

M2M - Machine to Machine

MAC - Media Access Control

MANET - Mobile Ad-hoc Network

MAP - Mesh Access Point

MBAP - Modbus Application Header

MID - Multiple Interface Declaration

MIMO - Multiple-Input and Multiple-Output

MIPS - Microprocessor without Interlocked Pipeline Stages

MP - Mesh Point

MPP - Mesh Portal

MPR - Multipoint Relays

NAT - Network Address Translation

NHDP - Neighborhood Discovery Protocol

NLOS - Non Line Of Sight

OBD - On Board Diagnostics

OBU - On Board Unit

OGM - Originator Message

OLSR - Optimized Link State Routing Protocol

OS - Operating System

OSI - Open Systems Interconnection

P2P - Peer-to-Peer

PDU - Protocol Data Unit

PID - Process Identifier

PIN - Personal Identification Number

PKI - Public Key Infrastructure

PL - Path Loss

RAM - Random-Access Memory

RERR - Route Error

RF - Radio Frequency

RFC - Request for Comments

RREP - Route Reply

RREQ - Route Request

RSSI - Received Signal Strength Indication

RSU - Roadside Unit

RTS - Request to Send

RTU - Remote Terminal Unit

Rx - Receiver

SAE - Simultaneous Authentication of Equals

SBC - Single Board Computer

SCADA - Supervisory Control And Data Acquisition

SCP - Secure Copy Protocol

SDK - Software Development Kit

SHA - Secure Hash Algorithm

SPI - Serial Peripheral Interface

SSH - Secure Shell

SSID - Service Set Identifier

STA - Station

TC - Topology Control

TCP - Transmission Control Protocol

TLV - Threshold Limit Value

TTL - Time-to-live

TTL - Transistor-Transistor Logic

Tx - Transmitter

UART - Universal asynchronous receiver/transmitter

UDP - User Datagram Protocol

USB - Universal Serial Bus

V2I - Vehicle-to-Infrastructure

V2R - Vehicle-to-Roadside

V2V - Vehicle-to-Vehicle

VANET - Vehicular Ad-hoc Network

WMN - Wireless Mesh Network

WPS - Wi-Fi Protected Setup

ZRP - Zone Routing Protocol

1. INTRODUCTION

1.1. CONTEXT

World population increased by 2.3 billion and is expected to be 67% more urban in 2050 [3]. With higher concentration in the cities comes the environmental concern. This dissertation arises in the context of the European project Future Cities intended to turn the city of Porto into an urban-scale living lab, where researchers and companies can test technologies, products and services, exploring subjects such as sustainable mobility, urban-scale sensing, as well as citizen's quality of life improvement. The Future Cities Project consists in three main platforms: (1) the Crowd Sensing platform, (2) the Vehicular network platform and (3) the Urban Sense platform. The Vehicular network, or *BusNet*, has vehicles equipped with On Board Units (OBUs) that offer free Wi-Fi to passengers and communicate between other vehicles within the *BusNet*. The Urban Sense platform is a composition of Data Collection Units (DCUs), each one comprising a set of sensors. The DCUs are scattered citywide in fixed positions as well as on-board buses making service public transportation, taxis and trucks. The *BusNet* allows the data collected from DCUs get to the Urban Sense cloud database (DB). This transport is achieved through Wi-Fi, Data Mulling or Global System for Mobile Communications (GSM).

This dissertation work is focused on creating a monitoring system for garbage street containers and create a network that extends the existent network to the remote garbage street containers. At present time the DCUs use the *BusNet* as a transportation medium. A possible solution is to use the IEEE 802.11b/g/n standard to make data from DCUs be delivered to the Urban Sense cloud DB through the public Access Points (APs) of Porto

city. Installing a GSM module in each node may be a simple solution from the installation point of view. However, due to the large amount of DCUs, the amount of data to be transferred and the monthly fee that telecommunications companies charge for that kind of services, make it costly unattractive as a global solution. Another solution is to opportunistically transfer DCUs data to the Urban Sense cloud DB through passing vehicles. Taking into account that the city of Porto has a vehicular network infrastructure already installed, this is an interesting solution.

1.2. MOTIVATION AND PROBLEM CHARACTERIZATION

This dissertation work aims to integrate the garbage street containers data in a Data Collection unit (DCU) using the Urban Sense Platform to monitor city wide garbage street containers. The DCU for Garbage Street Containers are sensing units that are composed by a Single Board Computer (SBC), a *HUB* and a sensor inside the garbage containers. The *HUB* is a proprietary device from *TNL* (a partner company), that communicates with the sensor inside garbage street containers. The *HUB* communicates with the sensors inside the garbage street containers via 433MHz radio frequency (RF) and at present time its sending data to their database (DB) via Global System for Mobile Communications (GSM) networks. This is an expensive solution and integrating this sensor into a DCU is a great opportunity to reduce costs and take advantage of the Vehicular Network and the Multi-hop network that extends the coverage to the remote garbage containers. To integrate this sensor in a DCU and make it communicate with the Single Board Computer, an interface had to be developed and the use of *Modbus* communication protocol had to be used. This is a restriction of the garbage street containers *HUB*. The Figure 1 depicts the vehicular network infrastructure and the coverage range of garbage street containers. Not every garbage street containers are within the coverage range of the *BusNet*. The yellow color represents the Bus lines. The green color represents the coverage criteria, around 67%. The red color represents the amount of garbage street containers that do not meet the criteria, around 33%. The criteria is, that every garbage street container has to be within a coverage radius of 50 meters at least 10s (consecutive) once or more per day.

Figure 1 - Garbage Street Containers Vehicular Network coverage

The Wi-Fi communications have coverage limitations. In Line Of Sight (LOS), the best it can offer are a few tens of meters. With all the buildings, vehicles and other aspects of an Urban Scenario this coverage can be very limited and susceptible to interference. There are many solutions to overcome this situation, from IEEE.802.15.4 (*ZigBee*) [4] to Long Range (LoRA) [5] communications for machine to machine (M2M) communication applied in the Internet of Things (IoT). However these technologies come with a cost. Take for instance the *ZigBee*, is a good solution, however it's not open source and due to its low power system the transmission distances can vary from 10 to 30m in LOS with transmission rates up to 250Kbit/s. It lacks on distance range and bandwidth. The LoRA is an interesting solution since it provides up to 21Km in LOS and 2Km in Non Line Of Sight (NLOS) with a Link budget up to $160\ dB$. Although such devices are also low powered they come with a very low bandwidth of less than 1Kbit/s. To cover the whole city of Porto, both technologies would require a great number of devices demanding for a huge investment. A GSM module is a simple solution but with a huge cost due to transmission fees.

This dissertation work aims to create independent DCUs for the garbage street containers that are not within the public Wi-Fi hotspots and the *BusNet* coverage. It also aims to implement a Multi-hop network to complement the actual *BusNet*, to make the data from this remote DCUs flows into the nearest vehicle from the *BusNet* or the nearest public Wi-Fi hotspots, scattered throughout the city of Porto. To provide real time information (Global Positioning System (GPS) and accelerometer data) from vehicles within the

BusNet, a TCP Server shall be developed. This server shall provide the On Board DCU with the information whether the vehicle is moving or stopped.

1.3. DISSERTATION OUTLINE

This dissertation is composed by seven chapters. In the chapter 1 is given the context, motivation and problem characterization. The chapter 2 describes the Future Cities Project with emphasis on 2 platforms, the Urban Sense Platform and the Vehicular Network platform (*BusNet*), as well as some related projects being developed by other organizations / companies. The chapter 3 describes the communication technologies used in this dissertation work. It's explained what are Ad-hoc links, Ad-hoc mode and *Wi-Fi Direct* as well as the difference between both technologies. The differences between Ad-hoc links and Multi-hop nerworks are shown. The routing protocols are explained as well as their elegibility and categorization. The IEEE 802.11s Hybrid Wireless Mesh Protocol (HWMP) is explained and there is also information regarding to routing metrics and security in descentralized networks. The *Modbus* protocol is explained with focus on the "*Modbus* RTU" and the holding registers function used in this dissertation work. The radio propagation models are described, focusing on the Free Space model and the Simplified Path Loss model. The chapter 4 describes the Wi-Fi Planner Tool (hardware and application) its functions and how it arose in the context of this dissertation work. The chapter 5 describes the proposed architecture and the implementation. It starts by showing the possible Single Board Computers that could be used in the Future Cities Project and describes the architecture of the Data Collection unit (DCU) for garbage street containers. The implementation of the DCU for garbage street containers is described, referring the RS485 physical layer, the *Modbus* protocol implementation and the DCU for garbage street containers application. The proposed Multi-hop network extension for DCU is described. The implementation of the Multi-hop network extension is described as well as the three methods to configure the Multi-hop network nodes, the script for field deployment and the setbacks of the implementation. The scripts for testing purposes are referred and the On Board Unit (OBU) application is described. The *TCP Server* developed for the OBU, the OBU limitations and the need to cross-compile are also described in this chapter. The chapter 6 shows the field experiments on Multi-hop networks and the two setups implemented to perform a coverage survey and performance tests. The chapter 7

presents the final conclusions with the respective contribuitions and results as well as the future work in this dissertation.

2. FUTURE CITIES PROJECT

The Future Cities Project is a project funded by the European Union (EU's) Seventh Framework Programme for Research (FP7) that brings together numerous partners, including universities and companies. The Porto Living Lab within the Future Cities Project is a set of three main platforms with a real impact in people's life. The three platforms are: the Crowd Sensing Platform, the Urban Sense Platform and the Vehicular Network Platform. These platforms are already implemented, but still continuously being improved and maintained.

The Crowd Sensing application, also known as SenseMyCity, runs on Android Operating System (OS) devices and enables the use of internal and external sensors in the smartphone to collect data from users. As an example of an external sensor that people can use with this application, is the use of a vital jacket - ambulatory Electrocardiogram (ECG) system, where the application running on the phone logs the ECG of the person wearing it. The analysis of data acquired with this kind of sensors, can lead to detection of heart diseases and city locations where people level of stress is higher. *SenseMyCity* application also gathers data from smartphone embedded sensors, such as Global Positioning System (GPS), magnetometer, accelerometer, gyroscope and supports Wi-Fi or Bluetooth devices for external sensors such as On Board Diagnostics (OBD) protocol to retrieve vehicles data, heart wave and beat sensor to retrieve user health. It has available several types of

algorithms for fuel consumption estimation, mobility pattern, traffic route similarity (carpooling) and Wi-Fi coverage maps. This provides to urban dwellers a mobility decision support and information about the city. This platform main targets of study are smart mobility and psychology field areas.

The Urban Sense platform includes a combination of several types of environmental sensors such as, image, noise, air quality, solar radiation, pollution levels, weather and meteorological sensors deployed city wide. This test bed collects various type of data, enabling applications and future research in the social sciences, urban planning, environment health and Information and Communication Technologies (ICT).

The Vehicular Network Platform, referred as *BusNet*, relies on a fleet consisting of 600 vehicles, including buses, taxis and trucks. In the core of this network are the data-mule system and opportunistic communications. The concept of data-mules lies in the data retrieval from sparse wireless sensor networks. This kind of communication is only available for equipment near the places where the vehicles stop frequently for the time needed to establish connection and data exchange.

2.1. URBAN SENSE PLATFORM

The Urban Sense is a platform consisting in a set of Data Collection Units (DCUs) and cloud database (DB) Servers. It includes 25 DCUs scattered city wide and 50 mobile DCUs installed on board city buses. The mobile DCUs can only take samples when the bus where they're installed is immobilized. Both types of DCUs gather data such as the number of people or vehicles in a restricted area, air quality, noise and meteorological conditions. The DCUs are equipped with a Wi-Fi interface 802.11b/g/n to send the data. The communication occurs when the DCUs successfully connect to the nearest public hotspot or the *BusNet*. When the DCU is connected to a hotspot, the data flows directly to the Urban Sense cloud DB. When the DCU connects to the *BusNet*, the data flows through the vehicles, using them as Data-mules and is delivered to the nearest hotspot or Roadside Unit (RSU).

The collected data allows identifying critical urban areas and evaluating the impact of urban intervention actions. This platform also enables the development of research and public interest projects in areas such as public health, urban transportation planning and environmental management. The goal of this platform is to understand and get aware of

environmental and behavior phenomena. In terms of impact for the city, aims to identify critical urban areas, detect events in real time and automatically, evaluate the impact of urban intervention actions. For the companies it allows to test business models, test products and proofs of concept. For researchers, this platform serves a variety of research fields, such as open data, big data, data analysis, wireless networks, data gathering, urban planning and transportation, climate, environment, health, visualization and learning tools.

2.1.1. DCUs Sensors

The Data Collection Units (DCUs) that Urban Sense Platform comprises include up to 13 specific sensors. These DCUs operate in specific places of the city, notice in Figure 2 that the DCU is embedded in a flower pot.

Figure 2 - DCU - Sensors (Real Picture from Rua das Flores)

The sensors in Figure 2 are fixed in a balcony over Rua Das Flores in Porto City. The sensor "A" is a wind vane, to measure wind direction. The sensor "B" is an anemometer, to measure wind speed. The sensor "C" is a pluviometer, to measure the amount of rain. The sensor "D" measures temperature and air humidity. The sensor "E" is a noise sensor, to measure the levels of noise in the street. The sensor "F" is a pyranometer and measures solar irradiance. The same type of fixed stations can be observed in Figure 3. In this picture the DCU is installed in a street semaphore.

Figure 3 - DCU - Sensors (Real Picture from Semaphores)

The data from these sensors is collected through these DCUs depicted in Figure 2 and Figure 3. The sensing units for the DCUs are divided in four groups of sensors, meteorological, air pollution, noise and video. The meteorological sensors group is composed by 75 units of humidity and temperature fixed and mobile sensors, 10 units of pluviometers, wind vane and anemometer fixed sensors, 75 units of luminosity fixed and mobile sensors and 10 units of solar radiation fixed sensors. The air pollution sensors group is composed by 75 units of Azote Dioxide fixed and mobile sensors, 75 units of Ozone (O3) fixed and mobile sensors, 50 units of Particles fixed and mobile sensors, 50 units of Carbon Dioxide fixed and mobile sensors, 50 units of Carbon Monoxide fixed and mobile sensors and 50 units of Total Suspended Particles fixed and mobile sensors. The noise sensors group is composed by 1 noise sensor for each DCU. The video sensors group is composed by 60 units of video cameras fixed and mobile and act as a sensor to count people. More DCUs are planned to integrate the Urban Sense Platform, such as sensors and controllers for garden watering systems, car parking and smart meters.

2.1.2. URBAN SENSE DCUS SOFTWARE ARCHITECTURE

The Data Collection Units (DCUs) are composed by a set of services running simultaneously. To better understand their software architecture, the Figure 4 depicts the four services running independently. (1) The Data Collector service is in charge of scheduling the polling to each sensor through the control board in order to obtain measurement values. (2) A local database (DB) where the obtained results are then stored. (3) A Data Sender service.

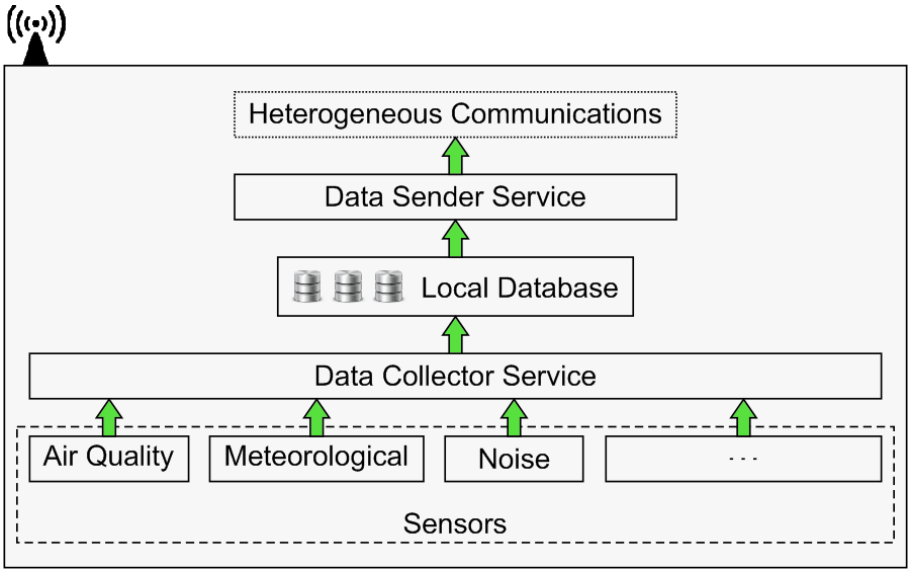

Figure 4 - Urban Sense Platform DCUs Software Architecture

The Data Sender Service is in charge of sending any measurements stored in the local DB to the central DB. To do that, the Data Sender Service communicates with another service running in a central server, rather than directly to the DB. This service performs authentication and basic data validation before accepting any data into the DB. The data is then stored in the central DB. If the data is successfully stored, an acknowledge message is sent to the data sender service. This service proceeds to the elimination of the corresponding samples from the local DB.

2.2. VEHICULAR NETWORK PLATFORM

The vehicular network test bed has already been installed on over 600 vehicles, 450 of which are buses, 100 taxis, 20 garbage trucks and 30 trucks located at Leixões harbor, representing the largest Vehicle-to-Vehicle (V2V) communication platform in the world, known to work at the time of this research. Each participating vehicle is equipped with an On Board Unit (OBU) capable to communicate through Wi-Fi, IEEE802.11p and Global System for Mobile Communications (GSM). The Porto city has 25 Roadside Units (RSUs) connected to high-speed optical network of Porto city. The RSUs are equipped with Wi-Fi and IEEE802.11p interfaces. As depicted in the Figure 5, the DCUs collect the data and send it through the *BusNet*. The data is then exchanged between vehicles through IEEE802.11p interface, until they are within coverage of a Wi-Fi Access Point (AP) or a RSU. When the data is delivered to a RSU or an AP, passes through validation

mechanisms and is stored in the Urban Sense cloud database (DB). An acknowledgment is sent back to the data originator to confirm that the data has reached the destination.

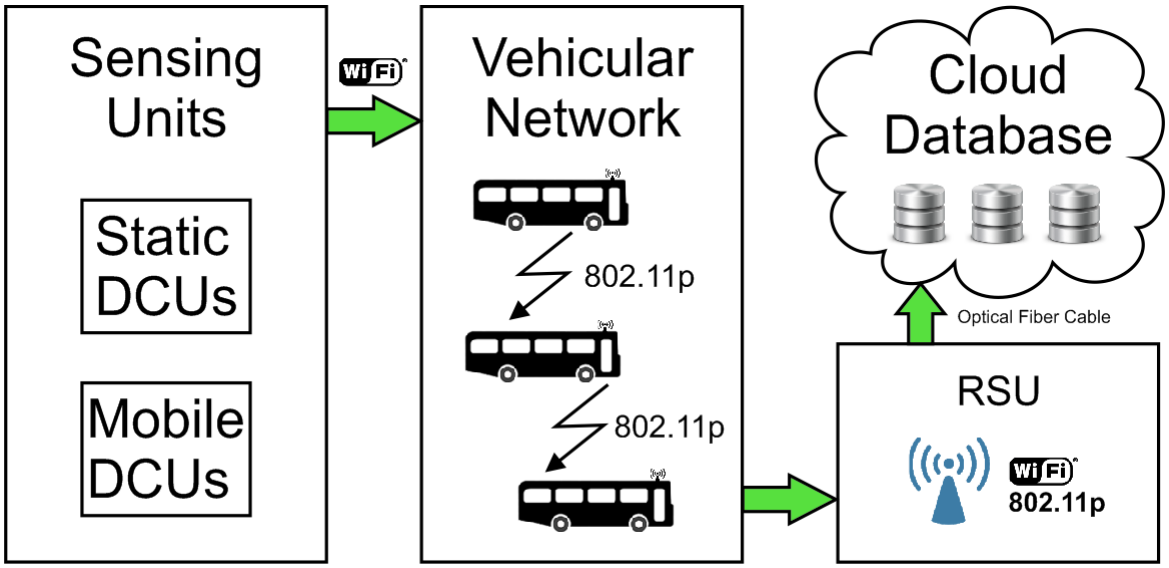

Figure 5 - Vehicular Network Architecture

Vehicular Networks often resort to Vehicular Ad-hoc Network (VANET) protocols, opportunistic networks and data-mulling [6][7]. In these challenging environments, popular Ad-hoc routing protocols such as Optimized Link State Routing Protocol (OLSR) or *Babel* fail to establish routes. This is due to these protocols trying to first establish a complete route and then forward the current data. However, when instantaneous end-to-end paths are difficult or impossible to establish, routing protocols must take a "store and forward" approach, where data is incrementally moved and stored throughout the network in hopes that it will eventually reach its destination. To fill this gap, there are works that aim to integrate the Delay and Disruption-Tolerant networks (DTNs) and the Ad-hoc On-Demand Distance Vector (AODV) routing protocol from Mobile Ad-hoc Network (MANET) [8]. This vehicular network uses specific network architectures and protocols such as DTNs. The DTNs, are characterized by their lack of connectivity, resulting in a lack of instantaneous end-to-end paths. The DTNs use the Bundle Protocol [9] and other routing protocols such as PRoPHET [10], Spray and Wait, Epidemic and Moby Space. The DTNs are based on the concept of data-mules. The DTNs can be implemented by using the IBR-DTN [11] API implementation [12][13]. When vehicles within a Vehicle Network are used to carry data, they are designated as data-mules.

The data-mules represented in the Figure 5 in form of buses, lies in the data retrieval from sparse wireless sensor networks. It consists of a three-tier architecture represented in

Figure 6, where the lower level consists in the sensor nodes that periodically perform data sampling from the surrounding environment. The middle level are the data mules, where they gather the data by moving around in the sensors covered area [14][15][16]. The higher level consists in a set of wired APs and data repositories that receive the data from the mules, where they can pass that information to a central database where the data is processed and stored.

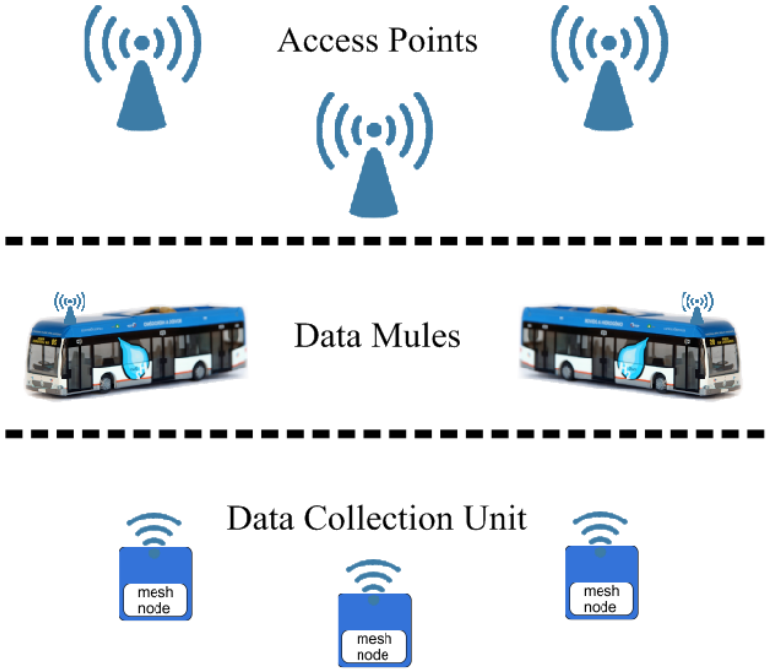

Figure 6 - Data Mules

Another approach within the scope of the Vehicular Network is the use of opportunistic networks. Opportunistic networks are one of the evolutions of MANETs [17]. In opportunistic networks, mobile nodes are able to communicate with each other even if a route connecting them never exists. Opportunistic networks are mobile wireless networks in which the presence of a "continuous" path between a sender and a destination is not assumed. Sender and destination nodes may never be connected to the network at the same time. The network is assumed to be highly dynamic, and the topology is thus extremely unstable and sometimes completely unpredictable. Nevertheless, the network must guarantee end-to-end delivery of messages despite frequent disconnections and partitions [18]. There are several protocols developed by many authors for opportunistic networks such as [19][20][21][22][23][24]. However in the context of this dissertation, this *BusNet* represents a transparent path from DCUs to the Urban Sense Platform cloud DB.

2.3. RELATED PROJECTS

Much like the Future Cities project, there are many other organizations/companies worldwide currently developing platforms and test beds like Urban Sense or Crowd Sensing.

The Citoyens Capteurs is a project proposed by citizen science association Citizen Lab - Citizens sensors. It takes place in Paris (France) and is carried out by an association that aims to put the skills of associates (Engineers and Researchers) to the service of civic, environmental and humanitarian causes. It aims to develop a citizen network of air pollution sensors available in open formats for their free use whether for scientific, civic and/or commercial purpose. This project allows creating a sensor network made from interconnected solutions of a wide range of sensors. The source code from the storage platform and the visualization from sensor measurements are available in the Citoyens Capteurs network [25].

The CITI-SENSE project is funded by the European Union (EU's) Seventh Framework Programme for research (FP7). It takes part in nine cities: Barcelona (Spain), Belgrade (Serbia), Edinburgh (UK), Haifa (Israel), Ljubljana (Slovenia), Oslo (Norway), Vienna (Austria), Ostrava-Bartovice (Czech Republic) and Vitoria-Gasteiz (Spain).The goal of the CITI-SENSE project is to give citizens the tools to "sense" their environment through new devices, such as smartphones, raising awareness of areas where pollution exists [26].

The EkoBus is framed within SmartSantander project. This project is funded by the EU through its Future Internet Research and Experimentation (FIRE) program. The EkoBus system has been deployed in the cities of Belgrade and Pancevo. The system uses the public transportation vehicles to monitor a set of environmental parameters over a large area as well as to provide additional information for the end-user like the location of the buses and estimated arrival times to bus stops [27].

The CitySens is a platform owned by the Urban Environment Observatory in Málaga (Spain). The CitySens is a platform for environmental monitoring based on Wireless Sensor Network embarked on public transportation vehicles. With this architecture it's possible to measure the concentration levels of polluting gases such as CO, NO2, ozone, etc. through this Wireless Sensor Network [28].

2.4. SUMMARY

This chapter describes the Future Cities Project and presents its three main platforms: the CrowdSensing, the Urban Sense platform and the Vehicular network platform. The CrowdSensing, also known as SenseMyCity it's an application that runs on Android OS devices to collect data from users. The Urban Sense platform is composed by a set of Data Collection Units (DCUs) that include a set of sensors measuring environmental parameters such as air pollution, meteorology and noise. The DCU sensors are described as well as the DCUs software architecture. The DCUs are deployed city wide in fixed spots as well as on board vehicles. The Vehicular network platform, referred as *BusNet*, relies on a fleet of 600 vehicles equipped with On Board Units (OBUs) capable to communicate through Wi-Fi, IEEE802.11p and 3G/GSM. The Porto city has 25 RSUs equipped with Wi-Fi and IEEE802.11p interfaces that communicate with vehicles. The DCUs are equipped with wireless interfaces that allow them to connect to any vehicle, RSU or AP. When DCUs collect the data, they send it through the *BusNet*, using vehicles as data-mules, nearby RSUs or nearby APs. In the third section are presented some of the projects being developed by other organizations/companies around the world that relate to the Future Cities Project, such as the Citoyens Capteurs, the CITI-SENSE project, the EkoBus and the CitySens.

3. COMMUNICATION TECHNOLOGIES

The communication technologies presented in this chapter are intended to give the reader an introduction to each one of them used in the context of this dissertation. In the first place, the operating modes referred in the IEEE 802.11 standard are explained with focus to the Ad-hoc operation mode. Then an introduction on how *Wi-Fi Direct* works and interacts between peers is presented. A comparison of the Ad-hoc mode and *Wi-Fi Direct* is then addressed. It's presented an overview of how the Multi-hop networks emerge through Ad-hoc links and routing protocols, as well as locating them in the Open Systems Interconnection (OSI) Reference Model. It's presented an introduction of the possible routing protocols that best fit this setup and how they are categorized. It's presented an overview of metrics in routing protocols and the security used in Multi-hop networks. The security issues are outside the scope of this dissertation work. Although there are some encryption systems, the difficulty of developing a secure system is a complex issue due to the absence of a centralized authority in this kind of networks. Lastly an introduction to the *Modbus* protocol focusing on the *Modbus* features used within the scope of this dissertation work is presented.

3.1. AD-HOC LINKS

Ad-hoc links can also be referred as Peer-to-Peer (P2P) wireless links. In wireless communications, the term P2P means that clients can talk directly to any other device that is within its radio range without relying on a pre-existent infrastructure, such as Access Points (APs) in infra-structured wireless networks.

3.1.1. IEEE 802.11 AD-HOC MODE

The IEEE 802.11 standard offers two basic modes of operation: infra-structure mode and the Ad-hoc mode. In infra-structured mode, wireless devices can only communicate with a central Access Point (AP). The AP is then responsible for re-transmitting packets from one client device to another client device, even if they are right next to each other. In Ad-hoc mode any device can start an Independent Basic Service Set (IBSS), and there is no hierarchy between IBSS devices, which is why usually an IBSS device is referred as a "node". When a node intends to join an IBSS network, it scans for a Service Set Identifier (SSID) to see if the network already exists, by passively listening on the channel and receiving beacons from other nodes or by sending probe requests. If an existing Basic Service Set Identification (BSSID) is detected, its BSSID is taken over and data frames can be exchanged directly between all IBSS nodes within radio reach. In this process of joining an IBSS, 12 frames are exchanged before current data traffic can be exchanged. However if encryption is not used, the data is exchanged right away, after receiving a probe response or beacon from the other node, as the authentication and probe request frames are optional. Depending on the setup of the Ad-hoc network, Dynamic Host Configuration Protocol (DHCP) can be used to provide Internet Protocol (IP) addresses. However, due to its distributed nature, Ad-hoc networks are usually configured with static IP addresses, whether these are IPv4 or IPv6 [29].

3.1.2. WI-FI DIRECT

Wi-Fi Direct is a Wi-Fi standard that enables devices to connect with each other easily without requiring an Access Point (AP). In a *Wi-Fi Direct* scenario there are 4 distinct stages before establishing a connection, these are the device discovery, provision discovery, group formation, and provisioning. After enabling the Peer-to-Peer (P2P) functionality of the Wi-Fi driver, a P2P device1 has to find out how many other P2P devices exist in its radio range. The purpose for the provision discovery is to get the Wi-Fi

Protected Setup (WPS) Personal Identification Number (PIN) Code or WPS push button. To start the group negotiation, one of the P2P devices will become a group owner (SoftAP) and the other P2P device will become an 802.11 client to connect to the SoftAP. After both P2P device1 and P2P device2 roles are confirmed, the P2P device which got the client role should launch the authentication service in the background and use the client authentication with the PIN code to perform the WPS procedure. In this process of establish communication, assuming that this is the most common case, at least 29 frames have to be exchanged before the first data transmission starts to flow. In the case of two devices re-invoking a persistent group, at least 18 frames are exchanged [29]. The P2P definition states "Peers are equally privileged, equipotent participants in the application" [30]. Taking this into account, *Wi-Fi Direct* is not a pure P2P protocol. It is essentially a protocol for forming hierarchical groups, and mostly used to connect just two devices. The devices are only equal peers until they connect to each other, and one of them becomes group owner. After that they follow the hierarchical roles of AP and client - master and slave. While the protocol, especially in combination with service discovery can provide a convenient and secure way to temporarily connect two devices in close proximity. On the other hand, Ad-hoc mode is a true P2P solution with no hierarchies where all participants are equal. This opens up a wide range of possibilities and communication modes, among them opportunistic, delay-tolerant networking and large-scale mesh networks, which are especially useful in scenarios where local infrastructure is untrusted or unavailable. Because IBSS mode is so simple and has many configuration options, it can be difficult to use. The IBSS mode starts to show its real strength over *Wi-Fi Direct* in combination with higher-layer protocols which implement IP allocation, service discovery, encryption and Multi-hop forwarding [30].

3.1.3. Wi-Fi Direct Versus Ad-hoc Networks

This section aims to give a general idea of the differences between *Wi-Fi Direct* and Ad-hoc networks as well as give a brief comparison between them, showing the advantages and/or disadvantages from each other. While *Wi-Fi Direct* is marketed as a replacement of Ad-hoc mode and is claimed to be "much more secure" and easier to set up, it does not cover all possibilities of Ad-hoc mode, most importantly the capability to form larger-scale networks and it comes at the cost of more complexity and only works well in limited situations.

The benefits of *Wi-Fi Direct* over Ad-hoc mode are:

- Easy to use for temporarily connecting a few devices;
- Security and encryption is built into the protocol;
- Service discovery is part of the protocol, although optional.

The drawbacks of *Wi-Fi Direct* over Ad-hoc mode are:

- Some topologies are not supported;
- Does not respond well to dynamic topology changes;
- Complex protocol;
- Many frames need to be exchanged before data traffic.

The benefits of Ad-hoc mode over *Wi-Fi Direct* are:

- Real Peer-to-Peer (P2P) solution;
- Capability to handle dynamic topology changes;
- Enables large-scale mesh networks;
- Simple protocol;
- No extra frames need to be exchanged before data traffic.

The drawbacks of Ad-hoc mode over *Wi-Fi Direct* are:

- Difficult to use for unexperienced users;
- All nodes in the network are on the same channel;
- No built-in security or encryption;
- No built-in service discovery.

Wi-Fi Direct and Ad-hoc mode solve very different use-cases and one cannot replace the other. *Wi-Fi Direct* is made for temporarily connecting a few devices in an easy-to-use and secure way, but it does not work well for larger network topologies. The Ad-hoc mode however is more versatile and difficult to set up, but can be used together with higher level protocols form large-scale mesh networks. To use the term peer-to-peer equally for both solutions is misleading since it leads to the assumption that the limited form of P2P of *Wi-Fi Direct* could substitute the real P2P capability of the Ad-hoc mode, which is not the case.

3.2. AD-HOC LINKS VS AD-HOC NETWORKS

Ad-hoc links get rid of the middle-man that is the Access Point (AP), however they don't have any inherent capability for Multi-hop. In the Figure 7 can be observed that if device A can reach device B, and device B can reach device C, but A cannot reach C, then A and C cannot communicate because B will not re-transmit any packets.

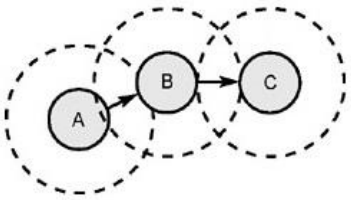

Figure 7 - Ad-hoc Nodes

Ad-hoc networks are very efficient, however they are limited to a hop distance, meaning that a device can only detect and communicate with another device within its radio range. The Ad-hoc mode does not allow Multi-hopping.

3.3. MULTI-HOP NETWORKS

Multi-hop networks are networks where the nodes communicate with each other using wireless channels and do not have the need for common infrastructure or centralized control such as Access Points (APs). In this wireless networks the devices usually operate in Ad-hoc mode and therefore referred as nodes. These nodes may cooperate with each other by forwarding or relaying each other's packets, possibly involving many intermediate relay nodes. This enables remote nodes that are not within their radio range to reach each other over intermediate relays without increasing transmission power. Such Multi-hop relaying is a solution for increasing throughput and providing coverage for large physical areas. This topology allows each node to act as a router and re-transmit packets on behalf of any other devices, providing the Multi-hop network the facility that Ad-hoc mode lacks. To better locate the routing protocols used it's presented the commonly used Open Systems Interconnection (OSI) Reference Model. The purpose of the OSI reference model depicted (left side) in the Figure 8 is to guide vendors and developers so the digital communication products and software programs created can interoperate and to facilitate clear comparisons among communications tools. The OSI allows referring an application to a specific layer depending on what it does.

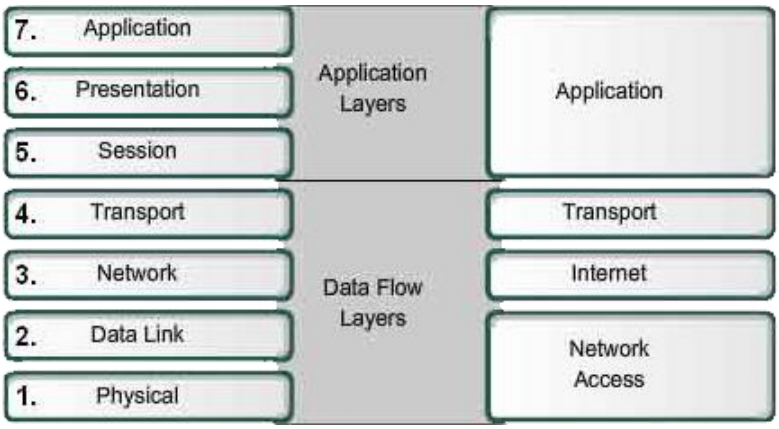

Figure 8 - Open Systems Interconnection (OSI) Reference Model

By combining Ad-hoc mode at OSI layer 2 and routing protocols above OSI layer 2 it's possible to create wireless Multi-hop networks purely between client devices without any need for centralized APs or routers. Several routing protocols have been developed to suit Ad-hoc networks. There are plenty routing protocols designed to solve specific problems. The routing protocols most suitable for this work are from Mobile Ad-hoc Network (MANET) and IEEE802.11. The MANET routing protocols run on layer 3. The routing protocols like the Hybrid Wireless Mesh Protocol (HWMP) and the WiFIX [31] do not respect the OSI Reference Model and therefore due to their features from layer 2 and layer 3, are referred as OSI layer 2.5 protocols. The routing protocols introduced in this dissertation work are MANET standard protocols defined by RFCs and the most common implementations for Multi-hop networks, with the exception for the HWMP. The HWMP is a Hybrid protocol that runs by default in the IEEE 802.11s extension to the IEEE 802.11 standard for mesh networks.

3.3.1. ROUTING PROTOCOLS ELIGIBILITY CRITERIA AND CLASSIFICATION

The routing protocols explored in the scope of this dissertation work are depicted in the Figure 9. They are categorized as Proactive (Table-Driven), Reactive (On-Demand) and Hybrid [32]. The routing protocols within each category follow the same main trunk. What makes them unique after this main trunk is their design and what they seek to solve / improve. As they get more specific features, they become unique.

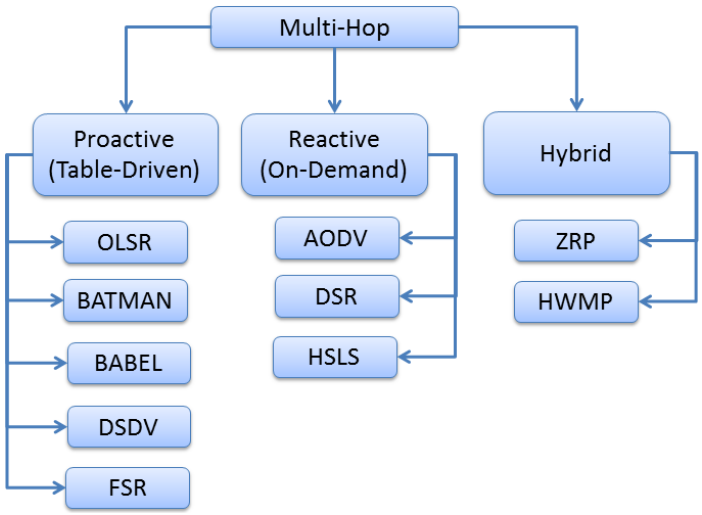

Figure 9 - Multi-hop Routing Protocols

Proactive (Table-Driven) routing protocols are based on the traditional Link State and Distance Vector algorithms, which were originally designed for wired networks. The routing information is stored in the structure of tables maintained by each node. These tables need to be updated due to frequent change in the topology of the network. The use of proactive routing algorithms allow mobile nodes to continuously evaluate routes to all reachable nodes and attempt to maintain consistent and up-to-date routing information regardless of whether data traffic exists or not. The Optimized Link State Routing Protocol (OLSR), Better Approach to Mobile Ad-hoc Networking (BATMAN), *Babel*, Destination-Sequenced Distance Vector (DSDV) and Fisheye State Routing (FSR) protocols are the examples.

Reactive (On-Demand) routing protocols involve discovering routes to other nodes only when they are needed. A route discovery process is invoked when a node wishes to communicate with another for which it has no route table entry. A route discovery operation invokes a route-determination procedure. The discovery procedure terminates either when a route has been found or there is no route available after examination of all route permutations. The Ad hoc On Demand Distance Vector (AODV), Dynamic Source Routing (DSR) and Hazy Sighted Link State (HSLS) protocols are the examples.

Hybrid routing protocols are a mixed design of the two approaches mentioned above. They combine merits of both the proactive and reactive approaches. Such hybrid protocols offer means to switch dynamically between proactive and reactive parts of protocol. For instance, proactive protocols could be used between networks and reactive protocols inside

the networks. The Zone Routing Protocol (ZRP) and Hybrid Wireless Mesh Protocol (HWMP) are the examples.

3.3.2. AODV

The Ad-hoc On-Demand Distance Vector (AODV) routing protocol, referred as RFC 3561 by IETF [33], is a reactive protocol for mobile Ad-hoc networks. The philosophy in AODV, like all reactive protocols, is that topology information is only transmitted by nodes on-demand. According Figure 10, when a node wishes to transmit traffic to a host to which it has no route, it generates a route request (RREQ) message that floods other nodes.

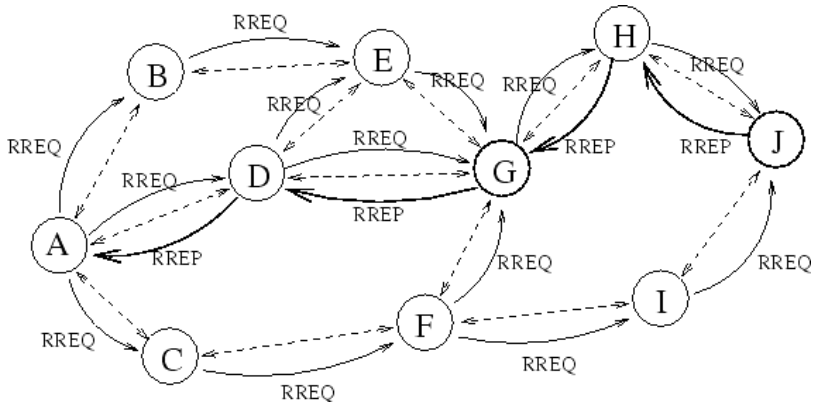

Figure 10 - AODV RREP between node A and node J [34]

In the example depicted in Figure 10, the node A initiates traffic to node J for which it has no route. Node A broadcasts a RREQ which is flooded to all nodes in the network. When this request is forwarded to J from H, J generates a route reply (RREP). This RREP is then unicasted back to node A using the cached entries in nodes H, G and D. This causes control traffic overhead to be dynamic and it will result in an initial delay when initiating such communication. A route is considered found when the RREQ message reaches either the destination itself, or an intermediate node with a valid route entry for the destination. For as long as a route exists between two endpoints, AODV remains passive. When the route becomes invalid or lost, AODV will again issue a request. AODV avoids the counting to infinity problem from the classical distance vector algorithm by using sequence numbers for every route. The counting to infinity problem is the situation where nodes update each other in a loop. AODV defines three types of control messages for route maintenance:

RREQ - A route request message is transmitted by a node requiring a route to a node. Every RREQ carries a Time-to-live (TTL) value that states for how many hops this message should be forwarded. This value is set to a predefined value at the first transmission and increased at retransmissions. Retransmissions occur if no replies are received.

RREP - A route reply message is unicasted back to the originator of a RREQ if the receiver is either the node using the requested address, or it has a valid route to the requested address. The reason one can unicast the message back, is that every route forwarding a RREQ caches a route back to the originator.

RERR – Route error message is used to notify other nodes of link loss. Network nodes monitor the link status of next hops in active routes. When a link breakage in an active route is detected, a RERR message is used to notify other nodes of the loss of the link. In order to enable this reporting mechanism, each node keeps a "precursor list", containing the Internet Protocol (IP) address for each its neighbors that are likely to use it as a next hop towards each destination.

The AODV routing protocol is designed for mobile Ad-hoc networks and can handle with low, moderate, and relatively high mobility rates, as well as a variety of data traffic levels. AODV is designed for use in networks where the nodes can all trust each other, either by using preconfigured keys, or because knowing that there are no malicious intruder nodes. AODV has been designed to reduce the dissemination of control traffic and eliminate overhead on data traffic, in order to improve scalability and performance.

3.3.3. OLSR

Optimized Link State Routing Protocol (OLSR), referred as RFC 3626 by IETF [35] is a proactive protocol for mobile Ad-hoc networks and is an optimization of the classical link state algorithm but designed for wireless networks. The OLSR routing protocol exchanges the following types of messages:

- "HELLO" messages are in charge of discovering the links between nodes, detect and signal the neighboring Multipoint Relays (MPR) nodes;
- Topology control (TC) messages are responsible for disseminating information about the topology of the network;

- Multiple interface declaration (MID) messages are responsible for declaring the presence of nodes with multiple interfaces;
- Host and Network Association (HNA) messages are responsible for providing a mechanism for injecting external information to OLSR networks domain.

The Figure 11 presents the following message exchange diagram between nodes, the MPR mechanism and OLSR control messages.

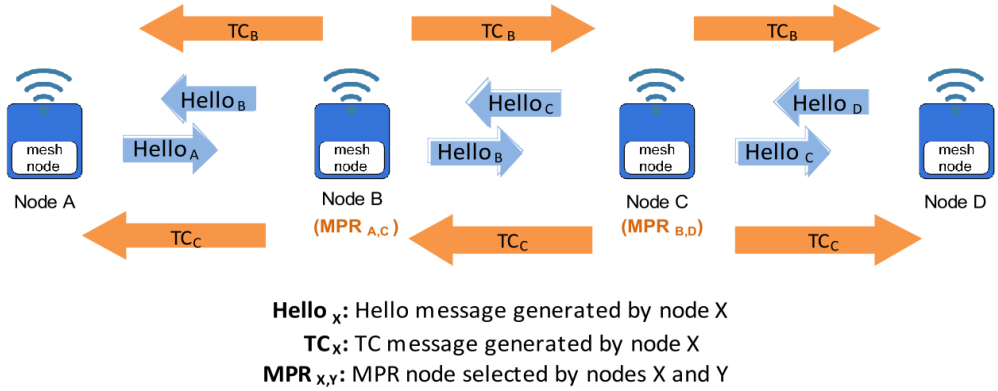

Figure 11 - MPR mechanism and OLSR control messages

The key point of OLSR is making use of MPR, which are nothing but nodes selected to forward packets on the network, which substantially reduces the required transmission and hence the traffic compared to other mechanisms, where all nodes forward the packets upon receiving them for the first time. OLSR is considered an efficient protocol for two main reasons, first because the information about the network status is only returned by selected MPRs and second, this protocol aims to minimize the number of control messages required between nodes. This protocol exchanges regularly information about the network topology to other nodes in the network. Each node selects a set of neighbor nodes as MPRs. The nodes which were selected as MPRs by some neighbor, report regularly through their control messages [36][37].

OLSRv2 [38] is the second version of OLSR and has been published by IETF on April 2014, referred as RFC 7181. OLSRv2 maintains many of the key features of the original version including MPR selection and dissemination. However the major differences are the flexibility and modular design using shared components such as packet format and the Neighborhood Discovery Protocol (NHDP). There are also differences regarding the handling of multiple address and interface enabled nodes in OLSRv2 comparing with the original version of OLSR.

3.3.4. BABEL

Babel [39] also referred as RFC 6126 by IETF is a proactive and loop-avoiding distance-vector routing protocol for IPv6 and IPv4 with fast convergence properties. It is based on the ideas of Destination-Sequenced Distance Vector (DSDV) [40], Ad-hoc On-Demand Distance Vector (AODV) [33] and Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP), it's designed to be robust and efficient both in networks using prefix-based routing and in networks using flat routing (mesh networks) and both in relatively stable wired networks and in highly dynamic wireless networks. The *Babel* routing protocol operates in a way that every *Babel* interface has an assigned router identity (ID), which is an arbitrary string of 8 octets that is assumed unique across the routing domain. By default router-IDs are assigned in the same manner as the IPv6 layer assigns host IDs. This protocol has two distinctive features:

The first is making a route selection considering the previous history to minimize the network impact when advertised as arriving to the destination via a route. In a situation where a node continually changes its preferred path between the source and destination can lead to an unstable routing. Therefore, when there is more than one route with similar link quality, route selection favors the previously established path instead of alternating between two routes.

The second is that *Babel* runs a reactive update, forcing a request for routing information when it detects a failure in any link of any of its favorite neighbors. Since the link quality measurements were previously completed in the initial stage, *Babel* claims to have a fast convergence time, routing almost immediately when a specific update is running.

The *Babel* messages (transmission and reception) are sent in the body of a User Datagram Protocol (UDP) datagram. The source address of a *Babel* packet is always a unicast address, link-local in the case of IPv6. Each *Babel* packet consists of a header followed by a sequence of Threshold Limit Values (TLVs). With the exception of Pad1 packets, all the TLVs follow a specific structure, (Type), (Length) and (Body). The messages in *Babel* routing protocol are the following TLVs:

- The "HELLO TLV" is used for neighbor discovery and for determining a link's reception cost.

- The I Heard You (IHU) TLV is used for confirming bidirectional reachability and carrying a link's transmission cost.
- The Router-ID TLV establishes a router-ID that is implied by subsequent Update TLVs.
- The Next Hop TLV establishes a next-hop address for a given address family (IPv4 or IPv6) that is implied by subsequent Update TLVs.

In normal operation, a *Babel* speaker sends both multicast and unicast packets to its neighbors. With the exception of "HELLO TLVs" and acknowledgements, all *Babel* TLVs can be sent to either unicast or multicast addresses, and their semantics does not depend on whether the destination was a unicast or multicast address. Hence, a *Babel* speaker does not need to determine the destination address of a packet that it receives in order to interpret it. To avoid collisions, a moderate amount of jitter is applied to packets sent by a *Babel* speaker: outgoing TLVs are buffered and should be sent with a small random delay. The *Babel* protocol has different tables for storing its own information and information from neighbors.

The neighbor table contains the list of all neighboring interfaces from which a *Babel* packet has been recently received. The neighbor table is indexed by pairs of the form (interface, address) and every neighbor table entry contains the following data: the local node's interface over which this neighbor is reachable, the address of the neighboring interface, a history of recently received "HELLO" packets from this neighbor, the "transmission cost" value from the last IHU packet received from this neighbor, the neighbor's expected "HELLO" sequence number. Note that the neighbor table is indexed by IP addresses, not by router-IDs: neighbourship is a relationship between interfaces, not between nodes. Therefore, two nodes with multiple interfaces can participate in multiple neighbourship relationships, a fairly common situation when wireless nodes with multiple radios are involved.

The interface table contains the list of interfaces on which the node speaks the *Babel* protocol. Every interface table entry contains the interface's "HELLO" *seqno* that is sent with each "HELLO" TLV on this interface and is incremented whenever a "HELLO" is sent. Note that an interface's "HELLO" *seqno* is unrelated to the node's *seqno*. There are two timers associated with each interface table entry, the "HELLO" timer, which governs the sending of periodic "HELLO" packets, the IHU packets and the update timer, which governs the sending of periodic route updates.

3.3.5. BATMAN

Better Approach to Mobile Ad-hoc Networking (BATMAN), referred as Internet-Draft by IETF [41] is a proactive protocol and was developed as an alternative to Optimized Link State Routing Protocol (OLSR). BATMAN provides a new approach for the route discovery that can be considered somewhere between the ideas of Ad-hoc On-Demand Distance Vector (AODV) and OLSR. The protocol procedure is to broadcast its own Originator Message (OGM). Each node periodically (OGM interval) generates a single OGM which is broadcasted on all interfaces. A jitter may be applied to avoid collisions.

BATMAN protocol doesn't try to find out the full routing path; instead it only learns which neighbor is the best gateway to each originator. It also keeps track of new originators and informs its neighbors about their existence. This protocol ensures that a route consists of bidirectional links only. On a regular basis every node broadcasts an OGM, thereby informing its neighbors about its existence. Neighbors who receive the OGMs relay them by broadcasting it. A BATMAN multi-hop network is therefore flooded with OGMs, as depicted in the Figure 12.

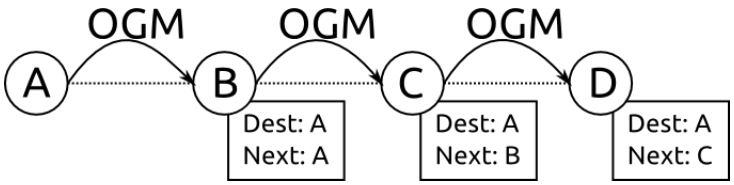

Figure 12 - BATMAN OGM Messages Exchange [42]

This flooding process is performed by single-hop neighbors in the second step, by two-hop neighbors in the third step, and so forth. OGMs are sent and repeated as User Datagram Protocol (UDP) broadcast messages, therefore OGMs are flooded until every node has received it at least once, or until they detect loss of communication links, or until their Time-to-live (TTL) value has expired. In practice OGM packet losses caused by interference, collision or congestion are significant. The number of OGMs received from a given originator via each neighbor is used to estimate the quality of a single-hop or multi-hop route. In order to be able to find the best route to a certain originator, BATMAN counts the originator-messages received and logs which neighbor relayed the message. Using this information BATMAN maintains a table with the best link-local router towards

every originator on the network. By using a sequence number, embedded in each OGM, this protocol can distinguish between new originator message packets and duplicates, ensuring that every OGM gets only counted once. BATMAN originators can announce themselves as gateways to the internet. Their announcement includes a gateway-class, which specifies the connection speed, uplink and downlink to the internet. Gateways also send a port-number which is used by gateway clients to establish a unidirectional UDP-tunnel to the gateway. The decision which gateway is selected for a destination is performed by the gateway-client. The method of tunneling between a BATMAN internet gateway client and the internet gateway ensures a stable route to the internet as long as the protocol can maintain a working communication path between both peers.

The BATMAN protocol was not designed to operate on stable and reliable media, such as cable networks, but rather to function on unreliable media inherently experiencing high levels of instability and data loss. The BATMAN protocol was devised to counteract the side effects of a network's fluctuation and compensate its instability, thus enabling high level of robustness. It also embodies the idea of collective intelligence opposed to link state routing. The topographical information is not handled by a single node, but spread across the whole network. No central entity knows all possible ways through the network. Every node only determines the data to choose the next hop, making the protocol very lightweight and quickly adapting to fluctuating network topologies.

3.3.6. IEEE 802.11s

As defined by IEEE, 802.11s is an extension to the IEEE 802.11 standard which allows multiple wireless nodes to connect with each other without having to connect to an Access Point (AP). The mesh topology allows direct communication between peers unlike the 802.11 standard that needs an AP. However, the real power of IEEE 802.11s manifests itself in the presence of multiple wireless nodes and by using the 802.11s mesh standard, the nodes can form a mesh network where all the links of the network are wireless. The IEEE 802.11s defines three kinds of nodes:

- Mesh Point (MP) – These node types support a Peer Link Management protocol, which is used to discover neighboring nodes and keep track of them, for communicating with other nodes that are farther than one hop from them, the MP supports the Hybrid Wireless Mesh Protocol (HWMP) [43].

- Mesh Portal (MPP) – These node types provide the users connected to the mesh network the access to the internet via these gateway nodes. These MPP are connected to both the mesh network and the internet, however these MPP must bridge at least two interfaces to provide the gateway functionality.
- Mesh Access Point (MAP) – These node types are like a traditional AP augmented with mesh functionality, so it can serves as an AP and be a part of the mesh network at the same time.

In a mesh network all nodes have frame forwarding capability and can thus forward frames originating at a node and destined for some other node. The IEEE 802.11s defines HWMP [43] as a default routing protocol, however the vendors allow the use of alternate protocols.

The Hybrid Wireless Mesh Protocol (HWMP) is used in MP and is considered a hybrid protocol because it supports two kinds of path selection protocols. Although these protocols are very similar to routing protocols IEEE 802.11s uses MAC addresses for "routing", instead of IP addresses and therefore, the term "path" is used instead of "route" and thus path selection instead of routing. This protocol is used for path selection which is necessary in order to find a "route" to a node which is not in the immediate neighborhood. The HWMP can be divided into two main parts, one part being a proactive portion which is basically a tree-based hierarchical routing protocol. The other part being an on-demand portion, which is a modification of AODV [33] routing protocol.

Within a mesh network, the MPPs operate as bridges. In the mesh, WLAN MPs need to keep an address table for path selection and frame forwarding. The IEEE802.11s provides the most advanced Wireless Mesh Network (WMN) concept in the LAN/MAN Standards Committee (LMSC). In its present form, IEEE802.11s covers all aspects of WMNs. Its usage scenarios foresee highly mobile applications whether in military or public safety users, enterprise networks and home environment. Due to its wide range of applications, several companies have announced future products to be compliant with IEEE802.11s [44].

3.3.7. ROUTING METRICS AND SECURITY

Each Multi-hop routing protocol has a specific algorithm and to determine the best path it takes into account the metric. Metric is a set of parameters and characteristics of wireless networks, such as variable link quality caused by noise or multi-path interference,

performance, and impact of routing traffic through multiple hops. Metrics are designed to provide ease of deployment and rapid convergence times while maintaining low channel overhead. Occasionally, a node in the network will become unavailable, due to disconnection or changes in the environment. Each node in the network constantly updates its routing tables with the optimal path to the network gateways. If the best path changes due to node failure or route metric, traffic will flow via the best known path. In case of a gateway failure or the emergence of a new gateway in the network with a better routing metric, all new traffic will be routed to the new gateway. Because certain gateways may be located on different IP subnets, each TCP flow is mapped to a particular gateway to avoid breaking established connections. The route through the network to the specified gateway may change over time, to adapt to network conditions.

The security issue in Multi-hop networks is still under development. There are security proposals for these topologies, since in Multi-hop or mesh networking the entities involved have no fixed roles to govern the behavior of participants, there is no initiator or responder, and there is no client or server. For instance, in Simultaneous Authentication of Equals (SAE) either party can initiate the conversation or both parties may initiate it simultaneously. This technique however is secure against passive attack, active attack, and dictionary attack [45]. The Security issue can be solved by protecting the Media Access Control (MAC) layer; encrypting everything, verification of authenticity and making sure the private keys are secure. Keeping too many keys and different security levels at every layer assists in solving security issues however Public Key Infrastructure (PKI) can be used. Asymmetric cryptography demands a lot from processing capabilities, making it expensive. In PKI, every node has a public/private key pair. The public part is made known to all other nodes, keeping the private key confidential. A Certificate Authority (CA) manages the key by distributing its own public key and signing certificate binding keys to nodes [46]. This method however implies a central entity, meaning that in a pure Mobile Ad-hoc Network (MANET) the network will be vulnerable since nodes can't get the certificate from third party. Regarding security there is also an interesting plugin for OLSR protocol, *SKiMPy*, this key management protocol allows a MANET node to agree on a symmetric shared key in the beginning of the network's lifetime to exchange digital certificates. The author of [47] sets a WMN test bed to secure OLSR by accessing different cryptographic approaches, such as *SKiMPy*, Internet Protocol Security (IPSec) among

others, where it concludes that the most efficient algorithms for authentication are Secure Hash Algorithm (SHA)-1 and AES.

3.4. *MODBUS* PROTOCOL

Modbus is an open protocol, meaning that it's free for manufacturers to build into their equipment without having to pay royalties. This protocol is often used to connect a supervisory computer with a remote terminal unit (RTU) in supervisory control and data acquisition (SCADA) systems. Versions of the *Modbus* protocol exist for serial lines ("*Modbus* RTU" and "*Modbus* ASCII") and for Ethernet (*Modbus TCP*). *Modbus* is an application layer messaging protocol, positioned at level 7 of the OSI model, Figure 8 p.22 which provide client/server communication between devices connected on different types of buses or networks. As depicted in the Figure 13, the *Modbus* protocol defines a simple Protocol Data Unit (PDU) independent of the underlying communication layers. The mapping of *Modbus* protocol on specific buses or network can introduce some additional fields on the Application Data Unit (ADU).

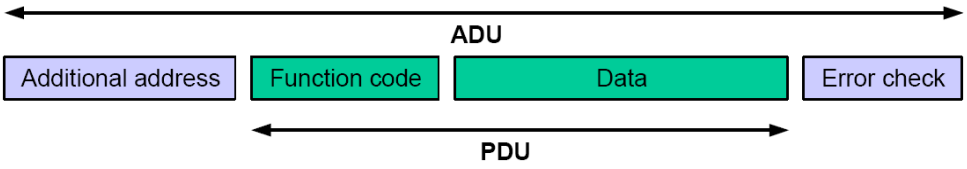

Figure 13 - General *Modbus* Frame [48]

The "*Modbus* ASCII" differs from the "*Modbus* RTU" in the message delimiting, the byte size, the way the data bytes are splitted and in the way it calculates the error check. In "*Modbus* RTU", bytes are sent consecutively with no space in between them and with a 3 and a half character space between messages for a delimiter. This allows the software to know when a new message is starting. Any delay between bytes will cause "*Modbus* RTU" to interpret it as the start of a new message. "*Modbus* ASCII" marks the start of each message with a colon character ":" (0x3A) and the end of each message terminated with the carriage return and line feed characters (0x0D and 0x0A). This allows the space between bytes to be variable making it suitable for transmission through some modems.

In "*Modbus* RTU" each byte is sent as a string of 8 binary characters framed with a start bit, and a stop bit, making each byte 10 bits. In "*Modbus* ASCII", the number of data bits is reduced from 8 to 7. A parity bit is added before the stop bit which keeps the actual byte size at 10 bits. In "*Modbus* ASCII", each data byte is split into the two bytes representing

the two ASCII characters in the Hexadecimal value. The range of data bytes in "*Modbus* RTU" can be any characters from 0x00 to 0xFF. The range of data bytes in "*Modbus* ASCII", represent only the 16 hexadecimal characters (between 0x0 and 0xF). Each "*Modbus* RTU" message is terminated with two error checking bytes called Cyclic Redundancy Check (CRC). Similarly, "*Modbus* ASCII" is terminated with an error checking byte called Longitudinal Redundancy Check (LRC). As a result, for the same message request, the "*Modbus* ASCII" takes roughly the double of the space it takes to the "*Modbus* RTU".

The *Modbus TCP* works with TCP/IP stack. These protocols are used together and are the transport protocol for the internet. When *Modbus* information is sent using these protocols, the data is passed to TCP where additional information is attached and given an IP address. The IP then places the data in a packet (or datagram) and transmits it. The TCP must establish a connection before transferring data, since it is a connection-based protocol. The Master (or Client in *Modbus TCP*) establishes a connection with the Slave (or Server). The Server waits for an incoming connection from the Client. Once a connection is established, the Server then responds to the queries from the Client until the client closes the connection. The "*Modbus* RTU" message is transmitted with a TCP/IP wrapper and sent over a network instead of serial lines. The Server does not have a Slave identity (ID) since it uses an IP address instead. Aside from the main differences between serial and network connections previously stated, there are a few differences in the message content. Starting with the "*Modbus* RTU" message and removing the Slave ID from the beginning and the CRC from the end results in the PDU. A new header in the *Modbus TCP* message emerges. Depicted in the Figure 14, a new 7-byte header called the *Modbus* Application Header (MBAP) header is added to the start of the message. This header contains the Transaction Identifier, which is a 2 byte set by the Client to uniquely identify each request. These bytes are echoed by the Server since its responses may not be received in the same order as the requests. The Protocol Identifier, that has 2 bytes set by the Client, always = 00 00. The Length that has 2 bytes identifying the number of bytes in the message to follow, and the Unit Identifier that has 1 byte set by the Client and echoed by the Server for identification of a remote slave connected on a serial line or on other buses.

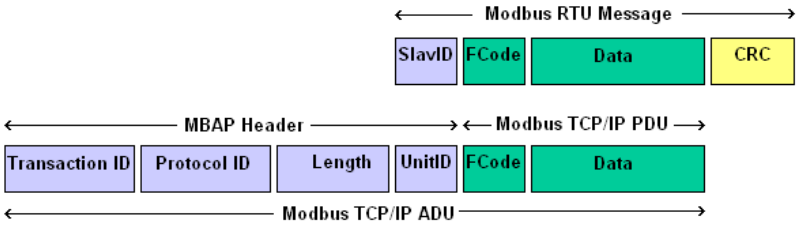

Figure 14 - TCP/IP ADU and *Modbus* RTU Message [49]

The size of the *Modbus* PDU is limited by the size constraint inherited from the first *Modbus* implementation on serial line network (max. RS485 ADU = 256 bytes). Therefore, the *Modbus* PDU for serial line communication = 256 - Server address (1 byte) - CRC (2 bytes) = 253 bytes. Consequently, the RS232 / RS485 ADU = 253 bytes + Server address (1 byte) + CRC (2 bytes) = 256 bytes. Therefore, the TCP *Modbus* ADU = 253 bytes + MBAP (7 bytes) = 260 bytes [48]. According the Table 1, the information is stored in the Slave device in four different tables. Two tables store on/off discrete values, called coils and two tables store numerical values called registers. The coils and registers each have a read-only table and read-write table. Each table has 9999 values. Each coil or contact is 1 bit size and assigned a data address between 0x0000 and 0x270E. Each register is 1 word size = 16 bits = 2 bytes and also has data address between 0x0000 and 0x270E.

Table 1 - *Modbus* Coil/Register Tables

Coil/Register Numbers	Data Adresses	Type	Table Name
1-9999	0x0000 to 0x270E	Read - Write	Discrete Output Coils
10001-19999	0x0000 to 0x270E	Read - Only	Discrete Input Contacts
30001-39999	0x0000 to 0x270E	Read - Only	Analog Input Registers
40001-49999	0x0000 to 0x270E	Read - Write	Analog Output Holding Registers

Coil/Register Numbers can be thought of as location names since they do not appear in the actual messages. The Data Addresses are used in the messages. For example, the first Holding Register, number 40001, has the Data Address 0000. The difference between these two values is the offset. Each table has a different offset. 1, 10001, 30001 and 40001. Each slave in a network is assigned a unique unit address from 1 to 247. When the Master device requests data, the first byte it sends is the Slave address. This way each slave knows after the first byte whether or not to ignore the message. The second byte sent by the Master device is the Function code. This number tells the slave which table to access and

whether to read from or write to the table. The most common messages used in the *Modbus* protocol are presented in the Table 2.

Table 2 - *Modbus* Function Codes

Function Code	Action	Table Name
01 (01 hex)	Read	Discrete Output Coils
05 (05 hex)	Write single	Discrete Output Coil
15 (0F hex)	Write multiple	Discrete Output Coils
02 (02 hex)	Read	Discrete Input Contacts
04 (04 hex)	Read	Analog Input Registers
03 (03 hex)	Read	Analog Output Holding Registers
06 (06 hex)	Write single	Analog Output Holding Register
16 (10 hex)	Write multiple	Analog Output Holding Registers

For this dissertation work, only one function was used, the "Read Holding Registers" function 0x03. This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at 0x00. Therefore registers numbered 1-16 are addressed as 0-15. The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits. The 2 bytes added to the end of every *Modbus* message are the Cyclic Redundancy Check (CRC), with the purpose of error detection. Every byte in the message is used to calculate the CRC. The receiving device also calculates the CRC and compares it to the CRC from the sending device. If even one bit in the message is received incorrectly, the CRCs will be different and an error will result. The Table 3 contains a *Modbus* request message as an example.

Table 3 - *Modbus* Request Message

0x40	0x03	0x00	0x00	0x00	0x02	0x2A	0xD2

The message request represented in the Table 3 is requesting the content of analog output holding registers 40000 to 40002 from the slave device with address 64 (0x40). The response message to the requested message from Table 3 is represented in the Table 4.

Table 4 - *Modbus* Response Message

0x40	0x03	0x06	0xA143	0x711A	0x236E	0x49	0xAC

The message in the Table 4 is the response of the message requested from Table 3, where the 0x40 is the Slave ID, the 0x03 is the Function Code (read Analog Output Holding Registers), the 0x06 is the number of data bytes to follow (3 registers x 2 bytes each = 6 bytes), the 0xA143 is the contents of register 40000, the 0x711A is the contents of register 40001 and the 0x236E is the contents of register 40002. The 0x49 and 0xAC are the last 2 bytes of the message which are the CRC.

3.5. RADIO PROPAGATION MODELS

Methods for predicting outdoor wireless signal coverage have been under development for decades. These models predict the signal power at a given point by determining the Path Loss (PL), the difference between the transmit power and received power, from the transmitter to the receiver. The Direct-Ray models are the models that calculate the PL signal based on parameters determined from the shortest straight line connecting the transmitting and receiving antennas. The simplest of these models is the purely distance dependent Friis transmission equation, which calculates the received signal power according to the signal loss in free space:

$$P_r = P_t \cdot G_r \cdot G_t \cdot \left(\frac{\lambda}{4 \cdot \pi \cdot d}\right)^2$$

Where the P_r is the power received, P_t is the power transmitted, G_r is the receiver antenna gain, the G_t is the transmitter antenna gain, λ is the wavelenght and d is the distance between the transmitter and the receiver as can be observed in the Figure 15.

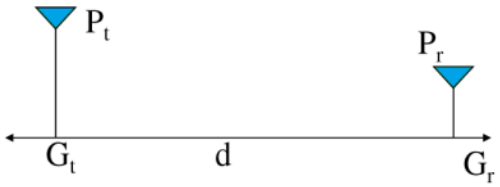

Figure 15 - Friis Free Space Equation Scheme

It may be remembered that Friis free space model is valid for d in the far field of the transmission antenna. The far field / Fraunhofer region is beyond the far field distance, where $d_f = \frac{2 \cdot D^2}{\lambda}$. It is related to the largest linear dimension of the antenna aperture and carrier wavelength, d is the largest linear distance of the antenna and D the dimension of

the physical length of the antenna or de diameter of the "dish" antenna. If the $d_f \gg d$ and $d_f \gg \lambda$ then it is the far field region. For path loss models d can't be 0. Therefore a close in distance is used which is known as the received power reference point. Thus $P_r(d)$ for $d > d_0$ may be reference to $P_r(d)$ may be predicted from Friis free space propagation loss model. It may also be obtained from measurements by using average of several recordings at distance d_0. The distance $d_0 \gg d_f$ but d_0 is sufficiently smaller than practical Transmitter (Tx) - Receiver (Rx) distance. Therefore the value d_0 in 1-2 GHz is approximately $1m$ for indoor conditions and from $100m$ to $1000m$ for outdoor conditions. For this purpose, the radio propagation models that were used, were the Simplified Path Loss model and the Free-Space propagation model [50].

3.5.1. FREE-SPACE PROPAGATION MODEL

The Free-Space propagation model is characterized by the equation [51, p. 29]:

$$P_G = \frac{P_r}{P_t} = \left(\frac{\sqrt{G_l} \cdot \lambda}{4 \cdot \pi \cdot d}\right)^2$$

Where P_G is the linear gain, the $\sqrt{G_l} = G_t \cdot G_r$ is the gain of the transmitter and receiver antennas and d is the distance between the transmitter and the receiver. The Free-Space propagation model can be also expressed in relation to a reference point, d_0.

$$P_r(\mathrm{d}) = P_t \cdot \mathrm{K} \cdot \left(\frac{d_0}{d}\right)^2 \qquad d \geq d_0$$

Where the P_r is the power received in a given distance, P_t is the power transmitted, K is a unit less constant that depends on the antenna characteristics and free space path loss up to distance d_0. The Figure 16 depicts where the distance d_0 stands as a reference distance between the Transmitter (Tx) and the Receiver (Rx) distance d.

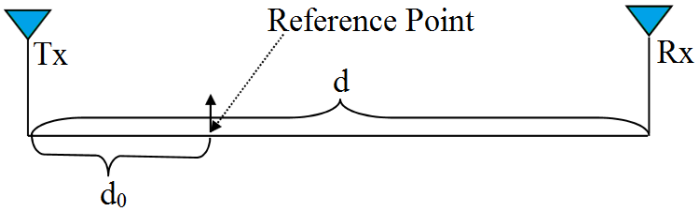

Figure 16 - Reference distance in Free-Space propagation model

This distance d_0 is the loss in signal strength of the electromagnetic wave in a Line of Sight (LOS) path through free space (usually air), with no obstacles nearby to cause reflection or diffraction.

3.5.2. SIMPLIFIED PATH LOSS MODEL

The Simplified Path Loss model is represented by the equation [51, p. 38]:

$$P_r = P_t \cdot \mathrm{K} \cdot \left(\frac{d_0}{d}\right)^{\gamma}$$

The path loss exponent represented by γ, can varies its value depending on the propagation environments. The value of the path loss exponent γ is 2 for the Free-Space model hence its use to define the reference distance. The value of the path loss exponent γ can also be set to 4, for the Two Rays model. The value of the path loss exponent γ can have different values depending on the environment. The Table 5 defines some general values for the most common environments. However the value of the path loss exponent γ can be obtained through field measurements.

Table 5 - Typical Path Loss Exponents [51, p. 39]

Environment	Path Loss exponent γ
Urban macrocells	3.7 to 6.5
Urban microcells	2.7 to 3.5
Office Building (same floor)	1.6 to 3.5
Office Building (multiple floors)	2 to 6
Store	1.8 to 2.2
Factory	1.6 to 3.3
Home	3

The important factor is to select the correct reference distance d_0. There are however some limitations in this model. Surrounding environmental clutter may be different for two locations having the same transmitter to receiver separation [50]. Moreover the received power predicted by path loss models is influenced by many factors such as reflection, diffraction, scattering and shadowing effects. For this experiment, were not considered atmospheric attenuation, multipath, absorption rainfall, fading, or cross Polarization, therefore the linear and the nonlinear PL in dB is as follows [50, p. 55]:

$$\mathrm{PL} = 10 \cdot \log\left(\frac{P_t}{P_r}\right) \Leftrightarrow \mathrm{PL}_{dB} = P_{t_{dB}} - P_{r_{dB}}$$

3.6. SUMMARY

In this chapter some of the communication technologies used in this dissertation work were described. The first section presented the Ad-hoc Links focusing on the Ad-hoc mode in the *IEEE 802.11* and the *Wi-Fi Direct*, showing their differences and limitations, benefits and drawbacks, then a small comparison showing that neither of them can replace one another, but rather serve different purposes. The second section presented the limitations of Ad-hoc mode comparing to Multi-hop networks, what distinguishes them and how they are used together. The third section described how Multi-hop network protocols are classified in the OSI reference model as well as their elegibility. The main features and messages exchanged from the following routing protocols: AODV, OLSR, *Babel*, BATMAN are described as well as HWMP, the mesh protocol from IEEE802.11s. The routing protocol metrics and security were also described to give an overview of what kind of encription systems exist for these kind of descentralized networks. The fourth section gave an overview of the *Modbus* protocol, focusing on the "*Modbus* RTU" and the read holding registers function that were used to communicate between the *HUB* and the Single Board Computer that compose the DCU for garbage street containers. The fifth section detailed the Radio Propagation Models such as Free-Space and Simplified Path Loss used to predict the Multi-hop nodes optimized position.

4. WI-FI PLANNER TOOL

In the scope of this dissertation work, the need to develop a field tool capable of planning a Multi-hop network based on the location and environment conditions. The Wi-Fi Planner Tool is a wireless planner tool developed to meet these requirements. This tool collects the Received Signal Strength Indication (RSSI) and Global Positioning System (GPS) information from a specific Service Set Identifier (SSID) broadcasted by the Multi-hop Network nodes. This tool is independent and mobile. To extend the capabilities of this tool, an external application is called. This application is the Kismet application [52]. This external application was used in the Wi-Fi Planner Tool to collect network Internet Protocol (IP) packets, the RSSI and GPS information from a SSID broadcasted by the Multi-hop network nodes to collect enough data to provide a wireless coverage map of the Multi-hop network. Kismet [52] is an 802.11 layer 2 wireless network detector and it can sniff 802.11b/g/n traffic. It works with any wireless interface capable to capture raw packets and support monitor mode. Kismet [52] identifies networks by passively collecting packets and detecting standard named networks. Kismet features the ability to log all sniffed packets and save them in a tcpdump/Wireshark compatible file format (.pcap). It also features the ability to detect default or "not configured" networks, probe requests, and determine what level of wireless encryption is used on a given Access Point (AP). In order to find as many networks as possible, Kismet [52] supports channel hopping. It also supports logging of the geographical coordinates of the network if the input from a GPS

receiver is additionally available. Kismet [52] consists of three separate components. (1) A drone that is used to collect packets, and then pass them on to a server for interpretation. (2) A server that can either be used in conjunction with a drone, or on its own, interpreting data packets, extrapolating wireless information and organizing it. (3) The client that communicates with the server and displays the information collected by the Kismet Server.

To install Kismet [52] in the raspberry Pi, the source code was downloaded and compiled for this platform. After compilation the configuration file was edited and the Wi-Fi interface and GPS were configured. After configuration, the Kismet Server [52] and the Kismet Client [52] were ready to run in the localhost.

The kisheat.py [53] is a tool that renders temperature-map style overlays for Google Earth application [2]. To get the heat map through the log files generated from Kismet [52], the kisheat.py [53] tool was installed. The kisheat.py [53] tool uses the log files generated by Kismet [52]. After kisheat.py [53] tool was executed, two things were generated, an "image.png" and a "Kismet.kml" file.

The Google Earth Application [2] was opened and the images generated along with the ".kml" files were loaded into the application that overlaid the images in the specific location provided by ".kml" files generating the heat map. The data collected through the use of this tool is used in this dissertation work to generate the heat map of the Multi-hop network nodes Wi-Fi coverage.

4.1. Wi-Fi Planner Tool Hardware

The Wi-Fi Planner Tool is a field tool, developed to gather a set of smaller tools in order to provide wireless Received Signal Strength Indication (RSSI) field observation measurements, Global Positioning System (GPS) coordinates, distance measurements and packets exchanged in the wireless network. This tool consists in a set of hardware components and a software component. The hardware components is composed by a GPS Receiver, a Wi-Fi interface, a Human Machine Interface (HMI) and a Battery as depicted in Figure 17. The software component is the Wi-Fi Planner Tool Application that does the measurements and calculations, providing an output.

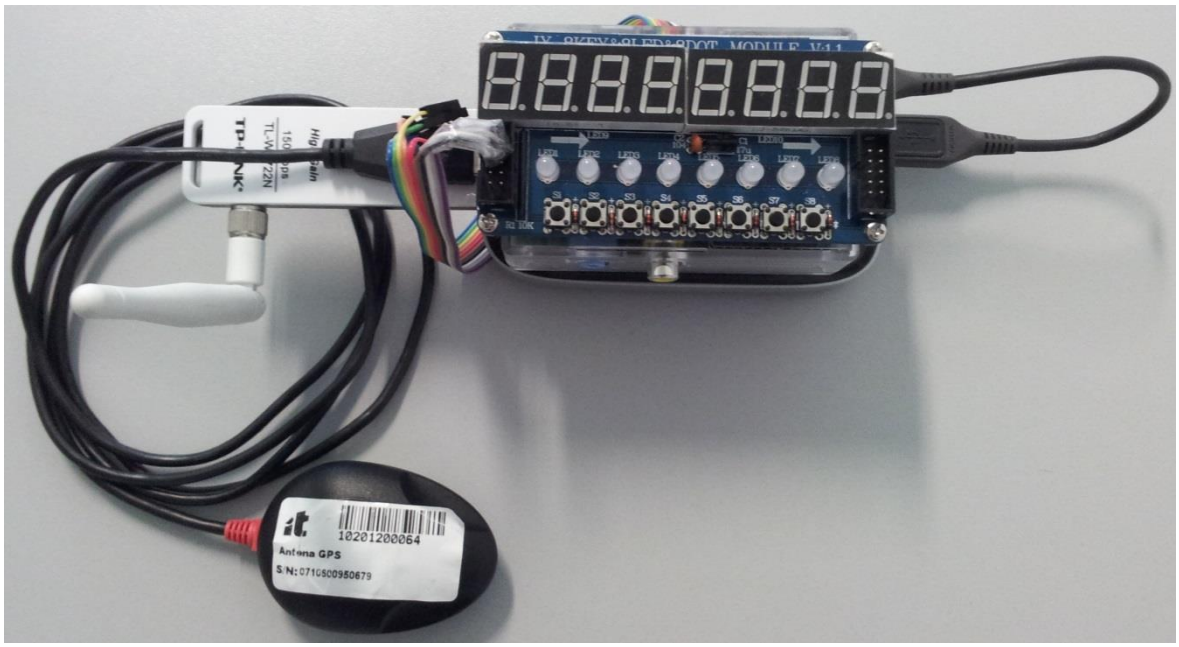

Figure 17 - Wi-Fi Planner Tool Hardware

The Wi-Fi Planner Tool hardware is divided in three main components. According the diagram in the Figure 18, this tool has a HMI which is an interface where the user can interact with the Wi-Fi Planner Tool Application. The Wi-Fi Planner Tool Application is the software component that runs in the Raspberry Pi platform. The GPS and the Wi-Fi are the external interfaces. The GPS receiver gets the geographic position coordinates and the Wi-Fi interface collects all data packets and RSSI.

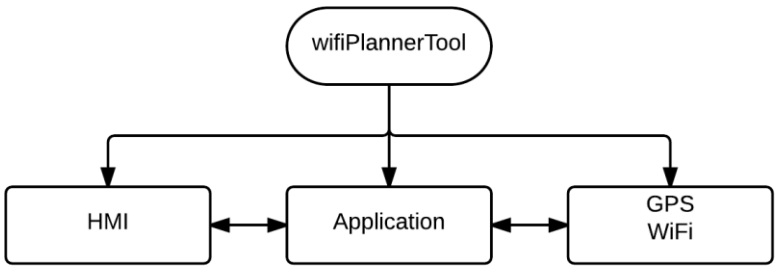

Figure 18 - Wi-Fi Planner Tool Hardware Diagram

The HMI used in the Wi-Fi Planner Tool is the TM1638 [54], a low cost and off the shelf part with 8 x 7 segment display, 8 bicolor Light-emitting diodes (LEDs) and 8 push buttons as depicted in the Figure 19. The HMI fits almost perfectly in the Raspberry Pi, allowing it to be used as a mobile tool for field measurements.

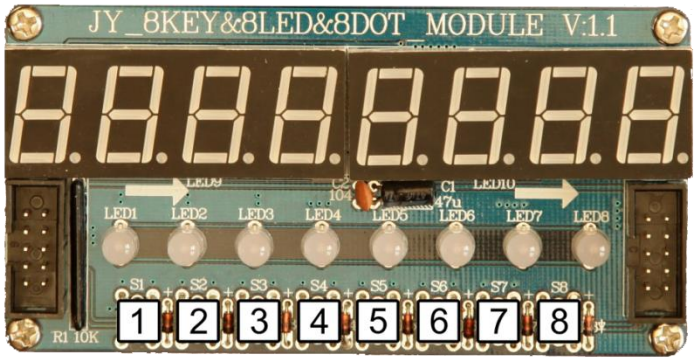

Figure 19 - Wi-Fi Planner Tool HMI

This device has a TM1638 [54] multiplexer and although the manufacturer does not provide a library for this device, it can be found in [55]. The tri state LEDs provide the user visual information of the processes state. The push buttons (from 1 to 8 in the Figure 19) are used to start and/or stop processes. The 7 segment displays are used to show results and alert the user with messages. The GPS receiver used in this dissertation work connects to the Raspberry Pi through a Universal Serial Bus (USB) port with a RS232 Transistor-Transistor Logic (TTL) circuit embedded, a *u-blox 5* GPS driver and a *GALILEO* Receiver. The Wi-Fi interface is a *TP-Link*, model *TL-WN722N* with a transmit power of 20 dBm and a $4dBi$ antenna gain, the same Wi-Fi interface used in the Multi-hop network nodes.

4.2. WI-FI PLANNER TOOL APPLICATION

The Wi-Fi Planner Tool Application is the software component, developed to operate in the Raspberry Pi platform. To develop the Wi-Fi Planner Tool the "BCM2835" and the "Pi-TM1638" libraries were installed previously in the Raspberry Pi. The Global Positioning System (GPS) receiver used in this dissertation work is "GPSd" compatible according [56]. Therefore the "GPSd" daemon package was also installed previously in the Raspberry Pi. The Wi-Fi Planner Tool Application is composed by five main functions as depicted in the diagram represented by Figure 20. The functions are categorized in the following manner: the Human Machine Interface (HMI), the Process Detection, the System Functions, the Measurement Functions and the Logging.

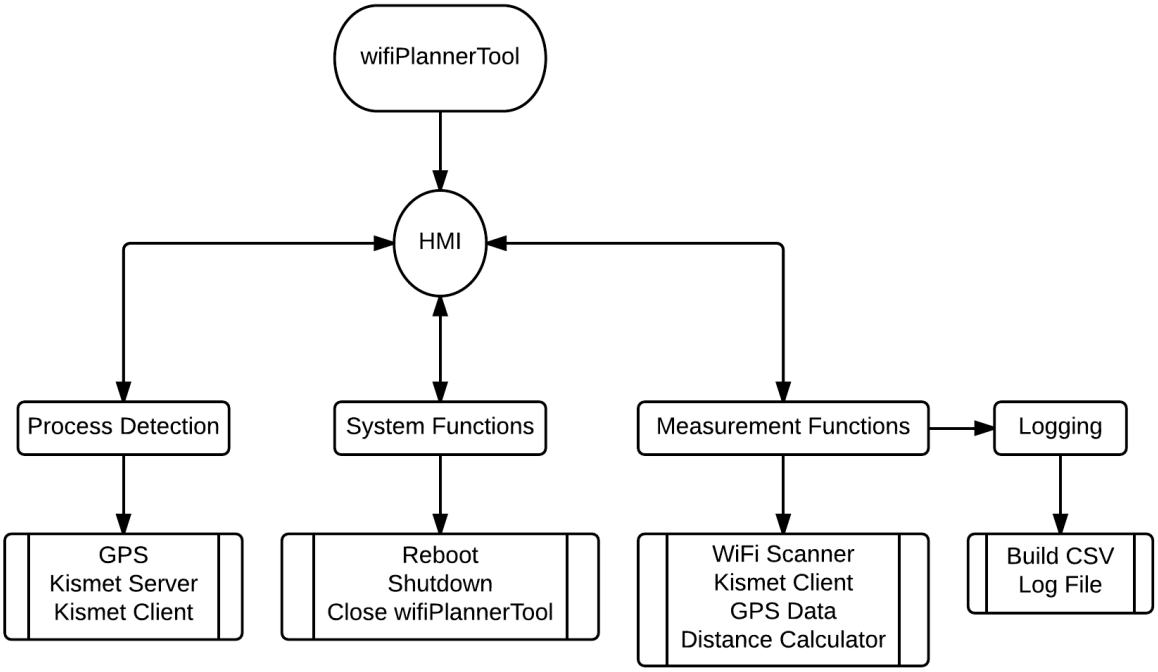

Figure 20 - Wi-Fi Planner Tool Application Overview

The Wi-Fi Planner Tool Application has a set of functions to detect processes states (active / inactive). These functions provide visual information of the GPS, Kismet Server and Kismet Client processes. The system functions are used to reboot, to shutdown the Raspberry Pi, and to close the Wi-Fi Planner Tool Application. The measurement functions are the Wi-Fi Scanner that scans for the Service Set Identifier (SSID) "mesh" (broadcasted by Multi-hop Network nodes) and returns the Received Signal Strength Indication (RSSI) in dBm, the Kismet Client is a function that calls the Kismet Client Application [52], the GPS Data is a function that gets the coordinates from the geographic location. The Distance Calculator is a function that based on the coordinates from GPS returns the distance in meters between the reference coordinates and the actual location.

4.2.1. HUMAN MACHINE INTERFACE

The Human Machine Interface (HMI) is where the user can interact with the Wi-Fi Planner Tool Application. The HMI starts by initiating the "BCM2835" and the "TM1638" libraries. The HMI upon boot informs the user with the message "START" as depicted in the Figure 21a, meaning that the application is ready to be used. The Wi-Fi Planner Tool application starts by entering in a loop that can be interrupted by the user.

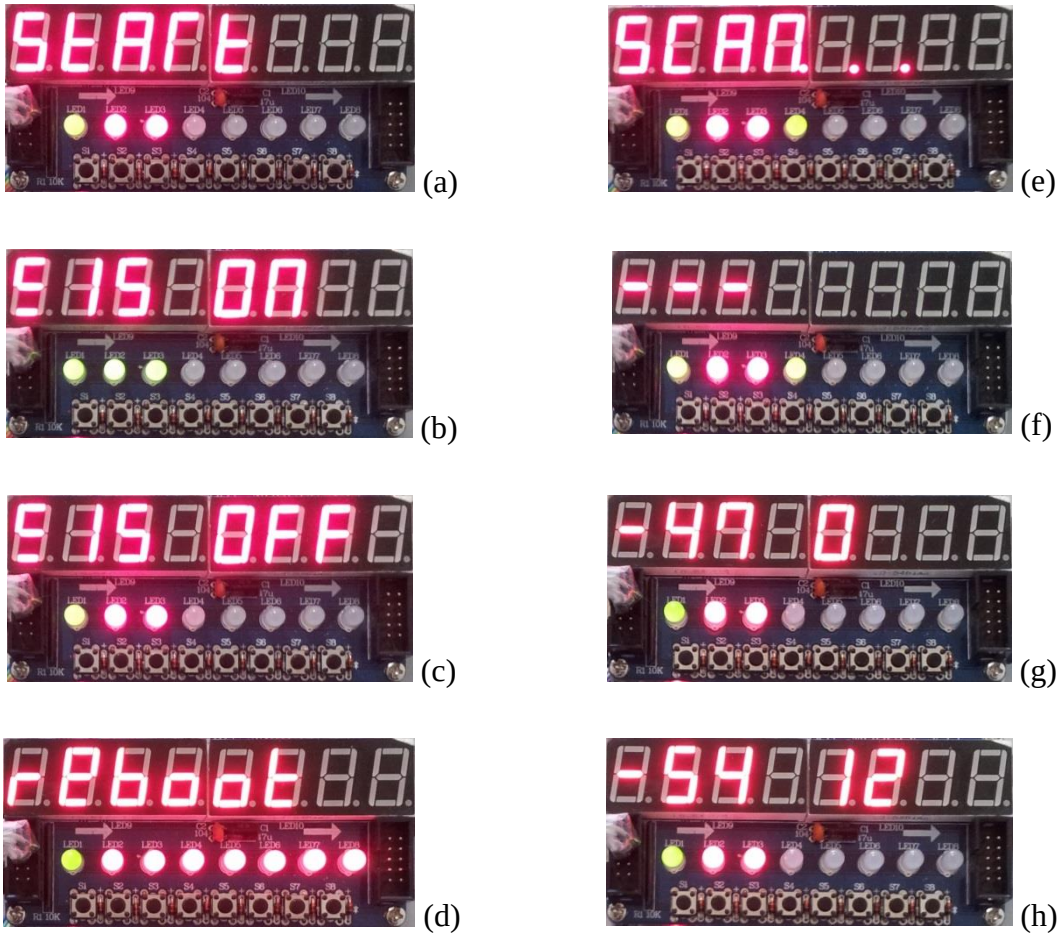

Figure 21 - Wi-Fi Planner Tool HMI

The Light-emitting diodes (LEDs) from the HMI have 3 states defined according the Table 6 and are used to alert the user of the state of each process that is being monitored.

Table 6 - Wi-Fi Planner Tool Application LED State

LED State	Description
Green	The process is running.
Red	The process has stopped.
Off	The process has ceased the operation.

On the Table 7 is shown what processes are being monitored and the corresponding LED that signals its state on the Wi-Fi Planner Tool Application.

Table 7 - Wi-Fi Planner Tool Application LEDs description

LED Nr.	Description
LED 1	Signals the GPS process.

LED 2	Signals the Kismet Server
LED 3	Signals the Kismet Client
LED 4	Signals the Wi-Fi Scanner

When the Global Positioning System (GPS) process is running the "LED 1" turns green, when the process is not running or the GPS Receiver is not connected the "LED 1" turns red, as shown in Figure 21a. In the Figure 21a the "LED 1" is green, this happens because the GPS is a process that starts on boot. The Kismet Server and Client are distinct processes and are detected using the same procedure used in the GPS, as depicted in the Figure 21b and Figure 21c. If both processes are valid (Figure 21b) the "LED 2" and/or the "LED 3" are green. The Kismet Server and Kismet Client are different processes due to the Kismet operation philosophy (One Server, many Clients). In this case the Server and the Client are used within the same device, therefore when Kismet is started, the Server starts first and the Client starts after. Hence their processes are detected accordingly.

The push buttons on the HMI (from 1 to 8 according the Figure 19) are used to start and/or stop processes. The Table 8 shows what processes are being controlled and the corresponding push button controlling it.

Table 8 - Wi-Fi Planner Tool Application Push buttons description

Buttons	Description
Button 1	Start the Kismet Server, Kismet Client and shows the message "KIS ON" in HMI, according the Figure 21b.
Button 2	Stop the Kismet Server, Kismet Client and shows the message "KIS OFF" in HMI, according the Figure 21c.
Button 3	Starts the Wi-Fi Scanner function, the GPS Data function, the Make CSV and logging functions, showing the message "SCAN…" according the Figure 21e.
Button 6	Turns off all features in HMI, it frees the memory and closes the Wi-Fi Planner Tool Application.
Button 7	Calls the function reboot, alerting the user through the HMI with a message "REBOOT" as depicted in the Figure 21d.
Button 8	Closes all the processes from Raspberry Pi and shuts it down.

The 7 segment displays on the HMI are used to inform the user with specific messages as well as the measurements being taken. The Table 9 shows what type of messages are shown to the user as well as their meaning.

Table 9 - Wi-Fi Planner Tool Application HMI Messages description

Messages	Description
Start	The Wi-Fi Planner Tool Application is Ready to be used.
KIS ON	The Kismet Server, Kismet Client are running.
KIS OFF	The Kismet Server, Kismet Client were stopped.
Reboot	Raspberry Pi is rebooting.
Scan…	The Wi-Fi Planner Tool is Scanning the Wi-Fi and GPS.
---	The Wi-Fi measurements are invalid.
RSSI | Distance	The RSSI is the Wi-Fi signal measured and the Distance is the distance to the reference point.

The flowchart in the Figure 22 refers to the Wi-Fi Planner Tool Application HMI.

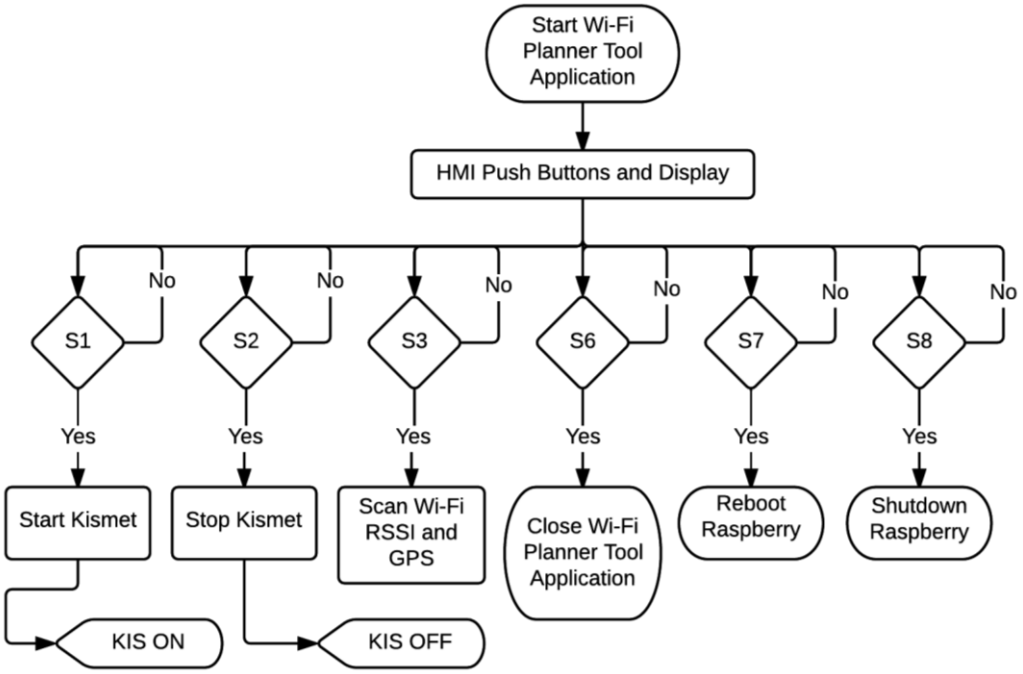

Figure 22 - Wi-Fi Planner Tool Application HMI and Push Buttons

When the Wi-Fi Planner Tool Application starts, enters in a loop that is constantly scanning the buttons state, calling the correspondent functions and showing information messages and measurements on the HMI.

4.2.2. PROCESS DETECTION

The Process Detection functions are functions that warn the user through the Human Machine Interface (HMI) Light-emitting diodes (LEDs) of processes that are running, not running or ceased. The process detection functions work with the same operating principle.

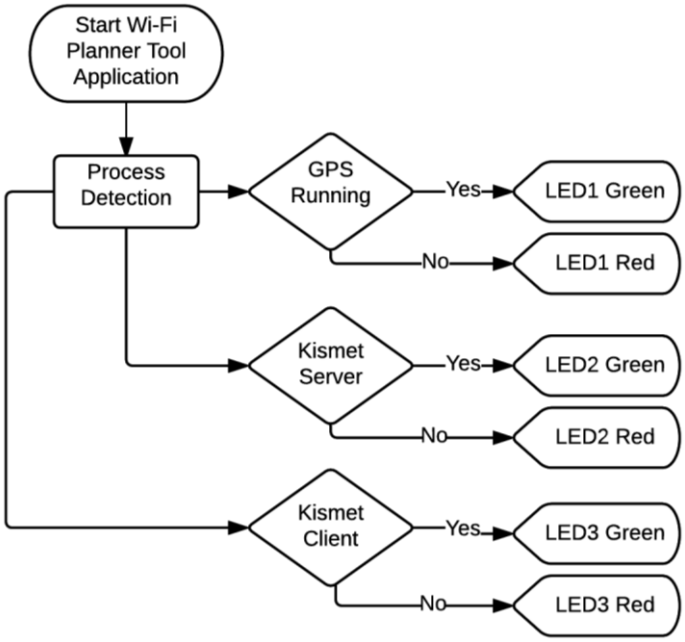

Figure 23 - Wi-Fi Planner Tool Application Process Detection

They start by creating a process and issuing the specific command to the system. The issued command returns the Process Identifier (PID) value for the selected process. If the function returned "0", then the process is not running. However if the function returned a value different from "0" it means that there is a valid PID for the issued process, meaning that the process is running.

4.2.3. SYSTEM FUNCTIONS

The System functions are used to perform system calls such as shutdown, reboot and close the Wi-Fi Planner Tool Application. The System functions follow the same operating principle. The Code excerpt 1 represents the reboot function from system functions.

```
int rpi_reboot() {
FILE * process;
char rpi[10];
process = popen("sudo reboot", "r");
memset(rpi,'\0',sizeof(rpi));
rpi[lastchar] = '\0';
lastchar = fread(rpi, 1, 10, process);
pclose(process);
return 0;
}
```

Code excerpt 1 - Reboot function

These functions start by creating a process and issuing the specific command to the system. In the above excerpt the reboot command was issued.

4.2.4. MEASUREMENT FUNCTIONS

The Measurement functions are a group of functions used to perform measurements and calculations, returning their output. The Wi-Fi Scanner Function scans for the Service Set Identifier (SSID) “mesh” broadcasted by Multi-hop network nodes displaying the Received Signal Strength Indication (RSSI) in dBm. The RSSI value shown in Human Machine Interface (HMI) is an arithmetic average computed from the 5 samples taken by default. This function starts by issuing an `iwlist` command with an `awk` script to parse the output. Based on the RSSI values this function computes the arithmetic average, the standard deviation and the variance (sample from population).

During the debug process on the Wi-Fi Scanner function. When the Wi-Fi Planner Tool is in Line Of Sight (LOS) (10m) or (100m) with a Multi-hop network node, the RSSI has little fluctuation (variance = 8 dBm^2), that could led to misinterpretation of the samples and the possibility to discard reliable data. When the signal is lost and the Wi-Fi interface starts to send the last value received, the variance is less than “1 dBm^2” in most of the situations. When the signal is lost and received again the variance reaches very high values. To prevent this situation, the variance value calculated presents a role of threshold to validate the measurements. When the Wi-Fi Planner Tool is being used and the variance is lower than “1 dBm^2”, this function discards that measure, sends the message “---“ according the Figure 21f to the HMI and notifies the user through Secure Shell (SSH) with detailed information.

The Figure 24 represents the flowchart from the Wi-Fi Scanner, the Global Positioning System (GPS) Data, the Distance Calculator, and the Logging functions.

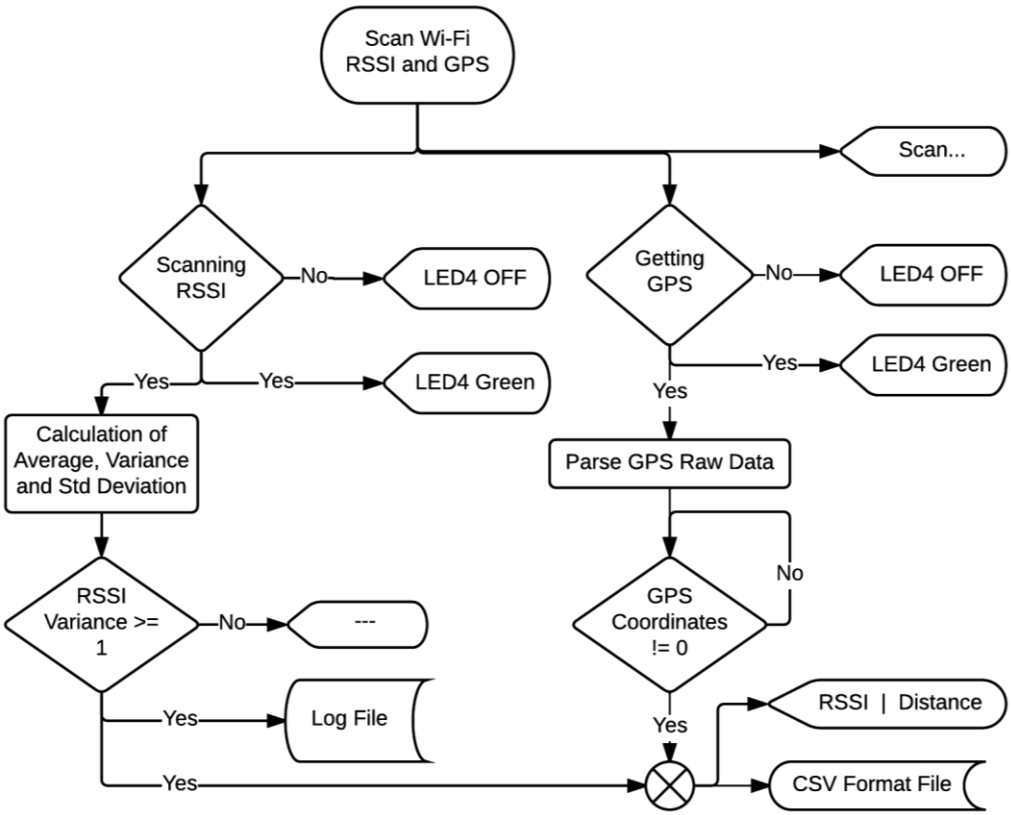

Figure 24 - Wi-Fi Planner Tool Application Measurement Functions

The GPS Data function is responsible to retrieve the latitude and longitude coordinates from GPS. This function starts by issuing a command line using `gpspipe`, an `awk` script and the `cut` tool, in order to parse only the information from latitude and longitude in the raw message outputted from GPS. The Code excerpt 2 from GPS Data function only shows how data is retrieved, converted and filtered. The source code of the Wi-Fi Planner Tool Application is available in the Appendix D.

```
float gps_data() {
…
do{
process = popen("gpspipe -w -n 5 | awk '/\"lat\":/{print $1}' | cut -d: -
f10-11", "r");
…
lastchar = fread(gpsdata, 1, 80, process);
strncpy(gps_lat, gpsdata, 12);
strncpy(gps_lon, &gpsdata[19], 12);
gpslatlon[0] = atof(gps_lat);
gpslatlon[1] = atof(gps_lon);
}while(gpslatlon[0] == 0 || gpslatlon[1] == 0);
pclose(process);
return *gpslatlon;
}
```

Code excerpt 2 - GPS Data function

The filtering part is due to the GPS "Startup" time, up to 42s. Until the GPS receive any information different from "0" in the latitude and longitude the user is notified through the HMI in the "LED 4".

The Calculate Distance is the function that is responsible to calculate the distance between geographic coordinates. This function uses a "flat earth" model [57] to calculate the distance between coordinates and returns the distance in meters. The "flat earth" model was used in this context, since the calculated distances are very short, therefore the error is considered insignificant. To calculate the distance between two geographic points, the current Federal Communications Commission (FCC) rules use two methods, the "flat-earth" method used in this application and the "spherical-earth" model [58]. The "spherical-earth" model uses a haversine [57] function which is a lot more complex and shall be used for very long distances which is not the case.

The Kismet Start function starts by opening a process and pass it the path to the `gpsKismet.start` bash script available in the Appendix N. To start Kismet [52] through the Wi-Fi Planner Tool, the script `gpsKismet.start` was developed. This script was built to only start the Kismet Server. To start the Kismet Client another process is created and the command is sent directly through the Kismet Start function to the system.

The Kismet Stop function starts by opening a process and send to it the path to the `gpsKismet.stop` bash script available in the Appendix N. To stop Kismet Server and Kismet Client [52] through the Wi-Fi Planner Tool, the script `gpsKismet.stop` was developed. It starts by verifying if the Kismet Client is running. If Kismet Client is running, stops the process, otherwise does nothing and performs the same procedure with the Kismet Server.

4.2.5. LOGGING

The Wi-Fi Planner Tool logs two types of files. A *Log* file ".log" with the purpose to be interpreted by humans and a Comma Separated Value (CSV) file (.csv) with the purpose to be used by applications capable to read a standard CSV file. The log file "WiFiPlannerTool.log" logs all the instant samples taken by the Wi-Fi Planner Tool as well as the computed values and adds a time stamp so that can be further analyzed. Although this function is ready to filter erroneous measurements, as a precaution measure

the lower limit is very low and the upper limit was not implemented in this version. The Code excerpt 3 from "WiFiPlannerTool.log" can be observed:

```
The instant measures are: -30 , -82 , -60 , -25 , -24 ,
Mesh Signal Level: -44 dBm , Std Deviation: 25.77 , Variance: 664.20 , Time:
Sat Feb  7 18:15:31 2015
```

Code excerpt 3 - Wi-Fi Planner Tool Log File output

The Code excerpt 3 is an output example from real measures taken, computed and logged by the Wi-Fi Scanner function.

The CSV file logs all the information retrieved and calculated by the Wi-Fi Planner Tool. It logs the Received Signal Strength Indication (RSSI), Standard Deviation, Variance, Reference Latitude, Reference Longitude, Latitude, Longitude, Distance and Time, according the CSV rules. The CSV file is organized by session, for each time the Wi-Fi Planner Tool is used it separates the measures taken while the application is operating. For each time the Wi-Fi Planner Tool is started it prints a header with the remaining measurements, as the example from Code excerpt 4 shows.

```
RSSI (dBm),Std Deviation,Variance,Reference Latitude,Reference Longitude,
Latitude,Longitude,Distance (m),Time
-58,27.32,746.30,41.179279327,-8.608565331,41.179279327,-8.608565331,0,Sat  Feb
7 18:12:50 2015
-41,24.18,584.80,41.179279327,-8.608565331,41.179279327,-8.608565331,0,Sat  Feb
7 18:13:12 2015
```

Code excerpt 4 - Wi-Fi Planner Tool CSV File output

The separation by session provides the user a visual separation from each time the Wi-Fi Planner Tool was used.

4.3. SUMMARY

This chapter describes the Wi-Fi Planner Tool. This field tool was developed to be independent, mobile and capable of planning a Multi-hop network, based on the location, environment conditions and resorting to the Radio Propagation Models. This tool collects the RSSI and GPS information from the SSID: "mesh" broadcasted by the Multi-hop network nodes. This tool also calls for an external application, the Kismet [52] Application (Server and Client). This external application generates log files that are used by the kisheat.py [53] tool. This is used to generate a heatmap of the Multi-hop network coverage range. The Wi-Fi Planner Tool is divided into two main components: the Hardware and the Application. The hardware is composed by a GPS Receiver, a Wi-Fi interface, a Human

Machine Interface (HMI) and a Battery. The software component is the Wi-Fi Planner Tool application that is divided into four groups of functions: the process detection functions, the system functions, the measurement functions and the logging functions. All these referred functions are called by the HMI function, which is the main function from the application. The process detection functions are used to detect the running processes such as GPS and Kismet server and/or client. The system functions are used to reboot shutdown and close the Wi-Fi Planner Tool application. The measurement functions are used to scan the RSSI the GPS location and calculate the distance from the reference point. The logging functions are used to log (.log and CSV) the values obtained for further analysis.

5. PROPOSED ARCHITECTURE AND IMPLEMENTATION

The proposed Multi-hop network architecture extends the *BusNet* existent architecture from Future Cities as a *BusNet* extension to the remote garbage street containers that are not within the coverage range of the *BusNet* and/or APs. The Data Collection units (DCUs) use public Access Points (APs) and use the *BusNet* to communicate with the Urban Sense cloud database (DB). The public APs are provided by *Porto Digital*, a public company of Porto municipality. Although the city of Porto has a wide variety of public APs, such as *Porto Digital*, *MEO*, *NOS*, or *Vodafone*, in this project only *Porto Digital* is being used due to the commercial nature of the other Wi-Fi networks and the existent partnership with Porto City Hall. At the time this dissertation work started, the Future Cities network architecture depicted in the Figure 25 allowed the DCUs, whether static or mobile, to communicate through public APs and/or the *BusNet*. The Multi-hop network proposed, extends the *BusNet* to the remote garbage street containers as well as provides a redundant path in areas where DCUs are within the coverage range of the *BusNet* and/or APs.

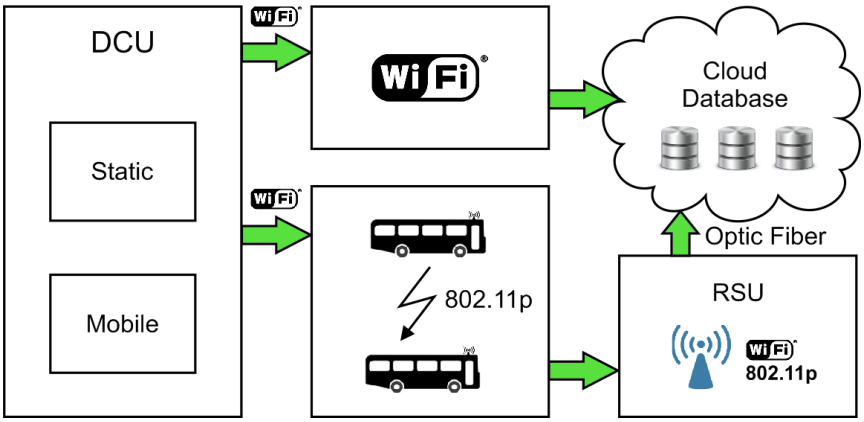

Figure 25 - Existent Network Architecture

When the data from DCUs is transported through the *BusNet* (802.11p), it is delivered to the Urban Sense cloud DB through the Roadside Units (RSUs). If the data from DCUs is transported through the Wi-Fi APs or through the Multi-hop Network, it is delivered directly to the Urban Sense cloud DB.

5.1. SINGLE BOARD COMPUTERS

Single Board Computers (SBCs) are used in the Future Cities project as a component of the DCUs. The SBCs include several computer functions in a single board, such as clock, microprocessor, Random-Access Memory (RAM), flash memory, Ethernet, Input/Output (I/O) control and sockets. The Future Cities project adopted the Raspberry Pi as the platform of choice for DCUs. The Table 10 presents nine characteristics from five platforms that could be used in the Future Cities project.

Table 10 - Single Board Computer Specifications

	Raspberry Pi Model B	**ODROID C1**	**BeagleBone Black**	**Cubieboard3**	**Intel Galileo**
CPU	ARM11 700MHz	A5(ARMv7) 1.5Ghz Quad Core	1GHz TI Sitara AM3359 ARM Cortex A8	1 GHz ARM Cortex-A7	400 MHz Intel Quark SoC X1000
GPU	VideoCore IV	Mali-450 MP2	3D Graphics Accelerator	Mali-400 MP2	N/A
RAM	512Mb	1Gb DDR3	512Mb	2Gb DDR3	256Mb DDR3
Connectors	2 x USB	4 x USB	2 x USB	2 x USB	1 x USB

	1 x Ethernet HDMI	1 x Ethernet uHDMI	1 x Ethernet uHDMI	1 x Ethernet HDMI	1 x Ethernet
Storage	SD Card	eMMC4.5 uSD Card	2 GB on-board eMMC uSD Card	8Gb NAND Flash uSD Card 1 x SATA	uSD Card
GPIO	17	28	65	54	14
Consumption	3.5W	10W	2.5W	10W	15W
Dimensions	85.60 mm X 53.98 mm	85 mm X 56 mm	86.36 mm X 54.61 mm	110 mm X 80 mm	100 mm X 70 mm
Price	27 €	27 €	43 €	82 €	75 €

The Raspberry Pi is a very balanced platform that suits the needs of the project, in terms of power consumption and General Purpose Input/Output (GPIO). Although the GPU from Raspberry Pi model B has the lowest performance compared to the others in Table 10 it was enough to the Future Cities Project. Although the Intel Galileo doesn't fit the needs for the Future Cities project due to the lack of Graphics Processing Unit (GPU), it was a board that was thought at the early beginning of the project. More information from Raspberry Pi compared to other platforms is available in [59] [60] [61] [62].

5.2. ARCHITECTURE OF THE DCU FOR GARBAGE STREET CONTAINERS

The Data Collection unit (DCU) for garbage street containers is composed by two distinct architectures: (1) a hardware architecture which includes a set of hardware components and (2) a software architecture where the network decisions are made and the data is stored in the local database (DB).

5.2.1. DCU FOR GARBAGE STREET CONTAINERS HARDWARE ARCHITECTURE

The proposed hardware architecture for the Data Collection units (DCUs) for garbage street containers as depicted in Figure 26 is composed by a Single Board Computer (SBC), an interface (RS232 to RS485) and the proprietary sensor from *TNL* (*HUB* and sensor inside the garbage street container).

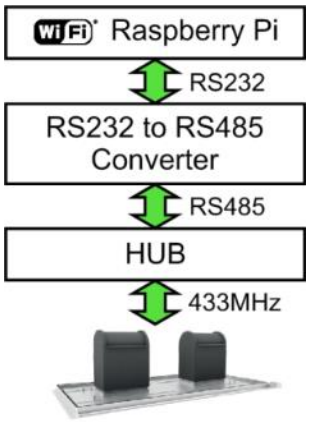

Figure 26 - DCU for Garbage Street Containers Hardware Architecture

According the hardware architecture depicted in Figure 26, the SBC used was the Raspberry Pi platform, used throughout the Future Cities Project. The Raspberry Pi is equipped with a Wi-Fi interface module, as depicted on the left side. The interface between the Raspberry Pi and the *HUB* is a serial communication converter from RS232 to RS485. This serial communication converter is a Transistor-Transistor Logic (TTL) interface that allows changes at the physical layer level. The *HUB* and the sensor inside the garbage street containers communicate with each other using a 433MHz radio frequency.

5.2.2. DCU FOR GARBAGE STREET CONTAINERS SOFTWARE ARCHITECTURE

The proposed software architecture for the Data Collection units (DCUs) for garbage street containers as depicted in the Figure 27 is the same software architecture used in all of the Future Cities Project DCUs. However the application developed to collect data is specific for the DCUs for garbage street containers.

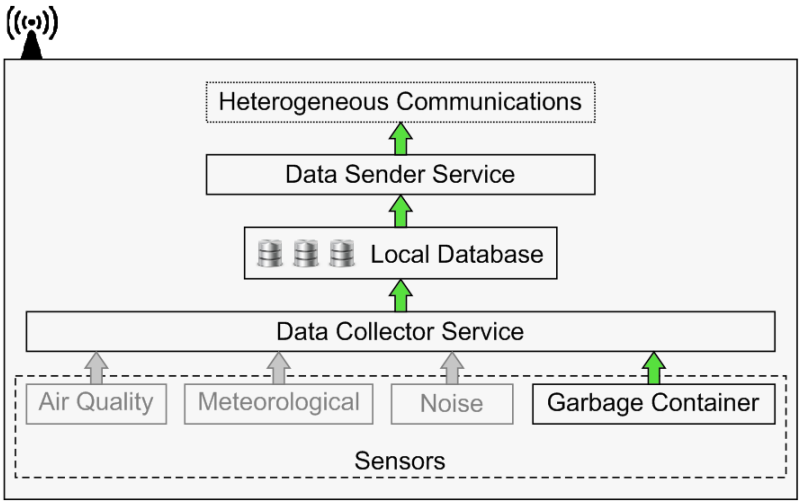

Figure 27 - DCU for Garbage Street Containers Software Architecture

The application developed to collect data from the *HUB* fits in the "Data Collector Service". This service is supported in the physical layer by the hardware architecture. In the software architecture is used the *Modbus* protocol to communicate between the *HUB* and the Single Board Computer (SBC). The *HUB* is responsible to collect the data from the sensor inside the garbage street containers. After the data is collected it's saved into a local database (DB) and then delivered to the Data Sender Service that is the responsible to decide which medium to use to send the data. The public Access Points (APs) are the preferred option to accomplish that, because of their stability, end-to-end connections and larger bandwidth. When they are available, sense units connect to them, establishing an end-to-end connection to the Urban Sense cloud DB and send their data. Taking into account the remote DCUs that are not covered neither from public APs or the *BusNet*, the data is sent through the proposed Multi-hop network.

5.3. IMPLEMENTATION OF THE DCU FOR GARBAGE STREET CONTAINERS

The Data Collection unit (DCU) for garbage street containers comprise a new set of sensors to collect data from containers filling levels, temperature, movements upon garbage collection, etc. This set of sensors, powered by batteries, is installed inside the garbage containers. The sensors communicate with a *HUB* through a 433 MHz low power proprietary wireless technology, both provided by *TNL*, the garbage containers manufacturer. The *HUB* receives data from several containers in the neighborhood and sends the collected data through General Packet Radio Service (GPRS) or through an embedded RS485 interface. The Urban Sense DCU for garbage street containers is based on the Raspberry Pi platform and uses its internal General Purpose Input/Output (GPIO) Universal asynchronous receiver/transmitter (UART) interface to connect to the *HUB* through a RS232 to RS485 converter. Depicted in the Figure 28 is the setup of a DCU for garbage street containers. Tagged with (1) is the Wi-Fi interface, with (2) is the Raspberry Pi platform, with (3) is the RS232 to RS485 converter used to interface with the *HUB*, finally tagged with (4) is the *HUB* provided by *TNL*.

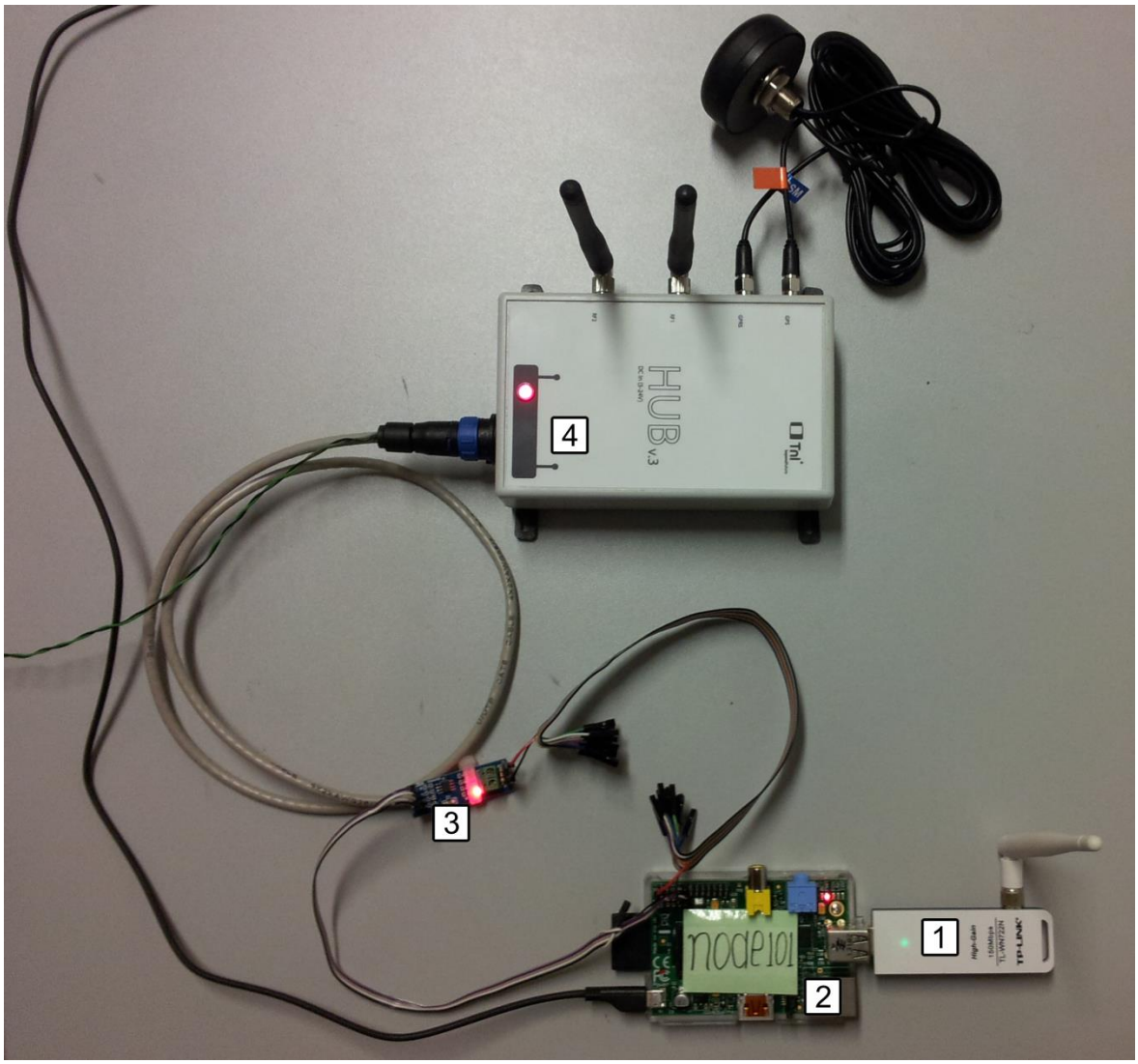

Figure 28 - DCU Garbage Street Containers setup

To collect the data from the *HUB* was developed an application to run in these DCUs. This application converts the raw data from the *HUB* to readable data and stores it in the local database (DB). The developed application communicates with the *HUB* by using the *Modbus* protocol, more specifically "*Modbus* RTU", over RS485 as a physical medium. The data collected is sent through the Wi-Fi interface using the same methodology as any other Urban Sense unit.

To communicate between the Raspberry Pi and the *HUB*, the RS232 to RS485 converter depicted in the Figure 30 was chosen. This solution is the cheapest (around 3€) and connects to the Raspberry Pi GPIO using only three Input/Output (I/O) pins. There are some solutions on the market to tackle this issue, such as small devices called "shields" that fit in the Raspberry Pi and provide extra functionalities to it. For RS232 and RS485

communications there is a shield called RasPiComm [63]. This shield has extra functionalities, however in this case none is needed with exception of the RS485. The disadvantages that this shield has are the price that is more expensive than the Raspberry Pi itself (around 40€) and the socket from the shield blocks any possibility for further use of other GPIO pins. For these reasons this shield was excluded. Another solution is a Universal Serial Bus (USB) to RS485 converter [64]. This solution has the advantage of the hardware drives the control line automatically and it's easier to implement. It's however expensive, around 25€, almost the Raspberry Pi price. This solution has the disadvantage (in this case) to use a USB port. Although the Raspberry Pi has 2 USB ports, none of them is available to this implementation work.

There are parameters and data from the garbage street containers that are set and defined by *TNL*. The time cycle between samples as well as the registers addresses and how Global Positioning System (GPS) coordinates are encoded were provided prior, to this implementation. The samples are taken with a time cycle of 4h, for exception of high temperature inside the container (possible fire), in this case a warning is sent to the *HUB*. Each time there is a refuse collection, the values are reset.

5.3.1. RS485 PHYSICAL LAYER

To communicate between the Raspberry Pi and the *HUB*, the company *TNL* provides a RS485 bus with physical media balanced interconnecting cable as a communication medium. The properties of differential signals provide high noise immunity and long distance capabilities. In the Data Collection unit (DCU) for garbage street containers the Raspberry Pi acts as a Master device within the *Modbus* protocol and the *HUB* act as a Slave device. In the Figure 29 is presented a half-duplex communication in RS485.

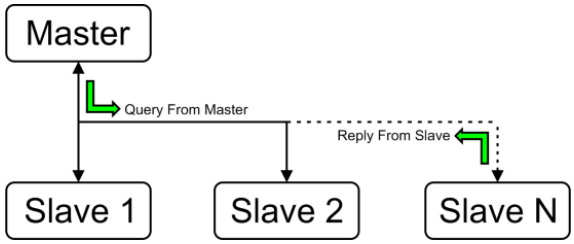

Figure 29 - Master / Slave Communication

In the RS485 physical medium, data is transmitted differentially on two wires twisted together, referred to as a "twisted pair". A 485 network can be configured in two ways,

“two-wire” or “four-wire”. In a “two-wire” network the transmitter and receiver of each device are connected to a twisted pair. “Four-wire” networks have one master port with the transmitter connected to each of the “slave” receivers on one twisted pair. The “slave” transmitters are all connected to the “master” receiver on a second twisted pair. In either configuration, devices are addressable, allowing each node to be communicated to independently. Only one device can drive the line at a time, so drivers must be put into a high-impedance mode (tri-state) when they are not in use. A consequence of tri-stating the drivers, is a delay between the end of a transmission and when the driver is tri-stated. This turn-around delay is an important part of a two-wire network because during that time no other transmissions can occur (not the case in a four-wire configuration). Some RS485 hardware handle this automatically, like the Universal Serial Bus (USB) to RS485 converter [64].

The Figure 30 depicts the RS485 to RS232 converter used to communicate between the *HUB* and the Raspberry pi in the DCU for garbage street containers

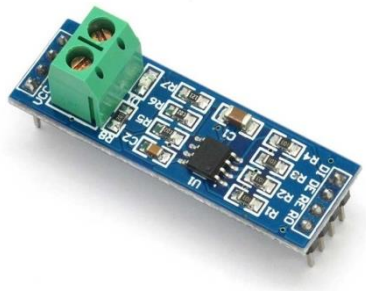

Figure 30 - RS232 to RS485 Converter

This RS232 to RS485 converter depicted in the Figure 30 has an on-board MAX485 chip. The MAX485 is a low-power and slew-rate-limited transceiver used for RS-485 communication. It works at a single “+5V” power supply and the rated current is 300µA. Adopting half-duplex communication to implement the function of converting Transistor-Transistor Logic (TTL) level into RS485 level, it can achieve a maximum transmission rate of 2.5Mbps. MAX485 transceiver draws supply current of between 120µA and 500µA under the unloaded or fully loaded conditions when the driver is disabled. The driver is limited for short-circuit current and the driver outputs can be placed at a high impedance state through the thermal shutdown circuit. The receiver input has a fail-safe feature that guarantees logic high output if the input is open circuit. In addition, it has a strong anti-

interference performance. The datasheet of the RS232 to RS485 converter is available in [65].

In the implementation of the DCU for garbage street containers, a "two-wire" topology was used since its only needed half-duplex communication due to the bus has very low traffic. In a conventional serial communication port (RS-232) the Request to Send (RTS) and Clear to Send (CTS) are handshake signals for flow control between machines, for use with half-duplex (one direction at a time). To emulate the serial port behavior, the RTS and CTS signals had to be bridged. In this case only the RTS signal is needed to handle the control line. The RS232 to RS485 converter depicted in the Figure 30 uses the control line ($\overline{RE}$ and DE bridged pins) to be handled by the DCU Application, using the RTS handshake.

Depicted on the Figure 31 is the electrical schematic from the DCU for garbage street containers. The "GPIO 17" is the RTS signal that had to be implemented in the Raspberry Pi serial communication "UART0" so it can be possible to capture the handsake signal.

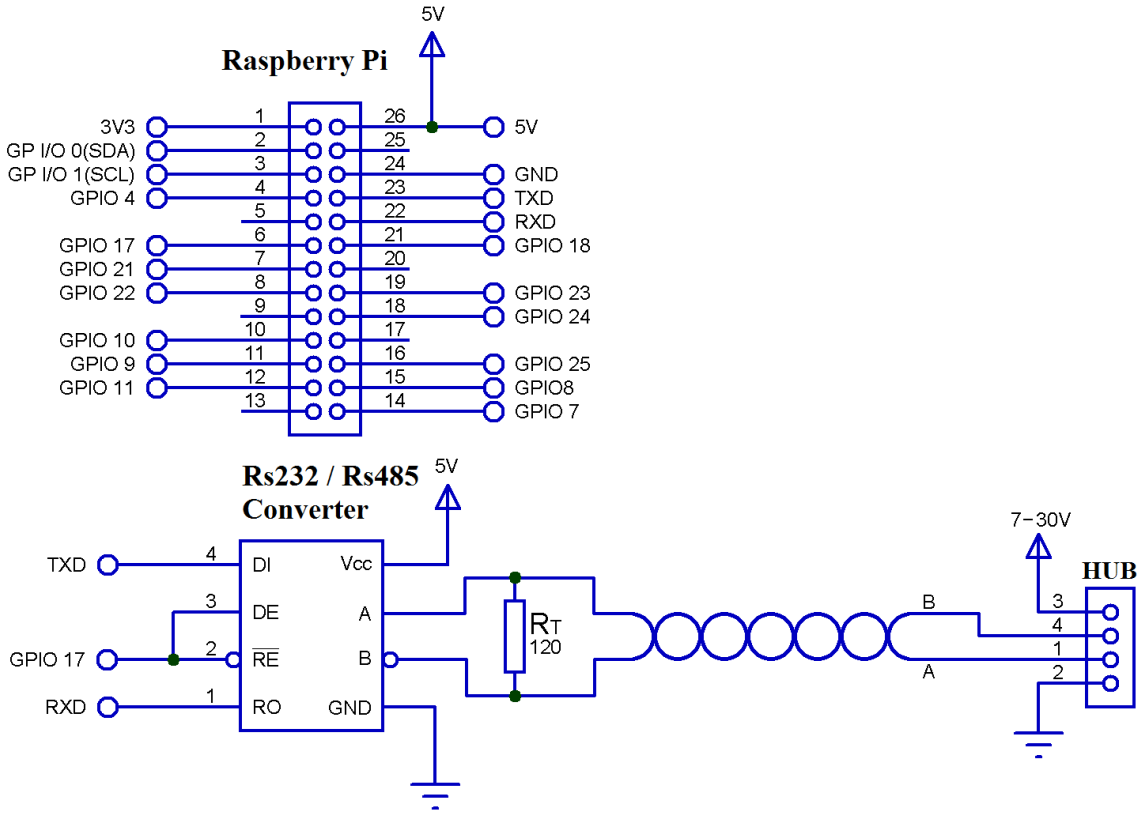

Figure 31 - DCU Garbage Street Containers Schematic

The Raspberry Pi uses a 5V power supply, the same voltage as the RS232 to RS485 converter. However the *HUB* needs an external power supply between 7V and 30V range, hence the need of an external power supply. The R_T is a terminator resistor. This 120Ω resistor has to be connected in one of the ends, for short distances in order to prevent signal "echoes" or reflections. In longer distances another R_T is needed in the opposite end. Usually and in this case, one R_T is on the Master device and the other R_T is implemented in each Slave device.

5.3.2. *MODBUS* PROTOCOL IMPLEMENTATION

In the communication between the Raspberry Pi and the *HUB* the *Modbus* communication protocol was used. To implement the *Modbus* protocol in the Raspberry Pi platform the BCM2835, the WiringPi [66] and the `libmodbus-3.1.2` [67] were installed. The BCM2835 is a library written in "C" language for the Raspberry Pi processor Broadcom *BCM 2835* Integrated Circuit (IC). It comes with several supports for Inter-Integrated Circuit (I^2C), Serial Peripheral Interface (SPI) and other features such as enabling the "GPIO 17" as a "CTS/RTS" signal to use as a control line in the RS485 communication emulating the serial port with the Universal Asynchronous Receiver-Transmitter (UART) serial communication. The WiringPi [66] developed by Gordon Henderson, is a General Purpose Input/Output (GPIO) access library written in "C" language for the BCM2835 used in the Raspberry Pi. The `libmodbus-3.1.2` is the *Modbus* communication protocol implementation for Raspberry Pi platform. By default the serial communication "UART0" from Raspberry Pi is used to debug the system boot. To operate with the "UART0" and prevent it to interfere with the RS232 to RS485 converter, the debug option had to be deactivated. To emulate the serial port for the Raspberry Pi and the Request to Send (RTS) signal for the RS232 to RS485 converter, the BCM2835 library was used. Despite the WiringPi library have the capability to emulate the RTS signal, in this implementation it was used for general purposes and debug.

With the *Modbus* communication protocol implemented in the Raspberry Pi and the RTS signal being emulated correctly, the development of the Data Collection unit (DCU) Application in p.65 started. During the development of the DCU for garbage street containers application were some problems to be tackled. The Master (DCU Application) was not detecting the Slave (*HUB*) address in the bus. This situation doesn't occur when the implementations meet the protocol specifications. However since the *Modbus* protocol

is widely used in industry, many companies modify some parameters in the protocol to meet their own requirements. To better understand what was happening in the bus, a logic analyzer was used to "sniff" the RS485 bus signals. The problem lied in the fast response from the *HUB* (Slave device) upon the Raspberry Pi (Master device) request. By the time the Master sent the request message and set the RTS signal down, the Slave was already replying. Therefore when the Master started to listen for the reply, it didn't receive the first bytes of the reply message, leading to a discarded message. To solve this issue, the RTS time had to be reduced in the `libmodbus-3.1.2` implementation of the Raspberry Pi. For this implementation the `modbus-rtu-private.h` library had to be changed in order to comply with the *Modbus* implementation from *TNL*. Because the *TNL Modbus* implementation differs from the implementation available to the Raspberry Pi platform, some adjustments had to be performed after the implementation took place. Inside the `libmodbus-3.1.2` library, in the file `modbus-rtu-private.h` is defined the timing for the RTS signal, set by default to 10000µs. The time that allowed for a steady communication between the Raspberry Pi and the *HUB* were 400µs. The Code excerpt 5 shows the line replaced in the `modbus-rtu-private.h` file.

```
#define _MODBUS_RTU_TIME_BETWEEN_RTS_SWITCH
400
```

Code excerpt 5 - *Modbus* library modification

This modification allowed to shorten the RTS signal time from 24,486ms (with 10000µs configuration), to 5,31ms (with a 400µs configuration). This modification allowed the Master to send the request message and set the RTS signal down fast enough to listen the reply from the Slave device.

5.3.3. DCU FOR GARBAGE STREET CONTAINERS APPLICATION

The application developed for the Data Collection unit (DCU) for garbage street containers runs in the Raspberry Pi platform. This application collects the data from the *HUB*, in a hexadecimal format, turning it into readable data and ready to be sent to the Urban Sense cloud database (DB). The *HUB* retrieves the data from the garbage containers and makes it available to the DCU Application through the *Modbus* protocol. As referred in the section 3.4 p.33, this application communicates with the *HUB* by using the "*Modbus* RTU" protocol and the RS485 referred in the p. 61 as a physical medium.

This application available in the Appendix B, starts by calling the function to start the BCM2835 General Purpose Input/Output (GPIO). The next function is the `setup_io()` function. This function represented in the Code excerpt 6 is responsible for open the shared memory region and mapping the GPIOs.

```
int main(int argc, char **argv)
{
bcm2835_init();
setup_io();
bcm2835_gpio_fsel(RPI_GPIO_P1_11, BCM2835_GPIO_FSEL_ALT3);
discover();
```

Code excerpt 6 - Raspberry Pi GPIO mapping function

The function `bcm2835_gpio_fsel()` represented in the Code excerpt 6 selects the pin 11 which is the "GPIO 17" on the first argument and on the second argument selects the alternative 3, which is the mask that enables the "RTS/CTS" function for RS485 communication. These steps are important due to the interoperability with the *Modbus* protocol, allowing the activation and deactivation of the Request to Send (RTS) signal. In the Figure 32 is shown the flowchart from the `discover()` function.

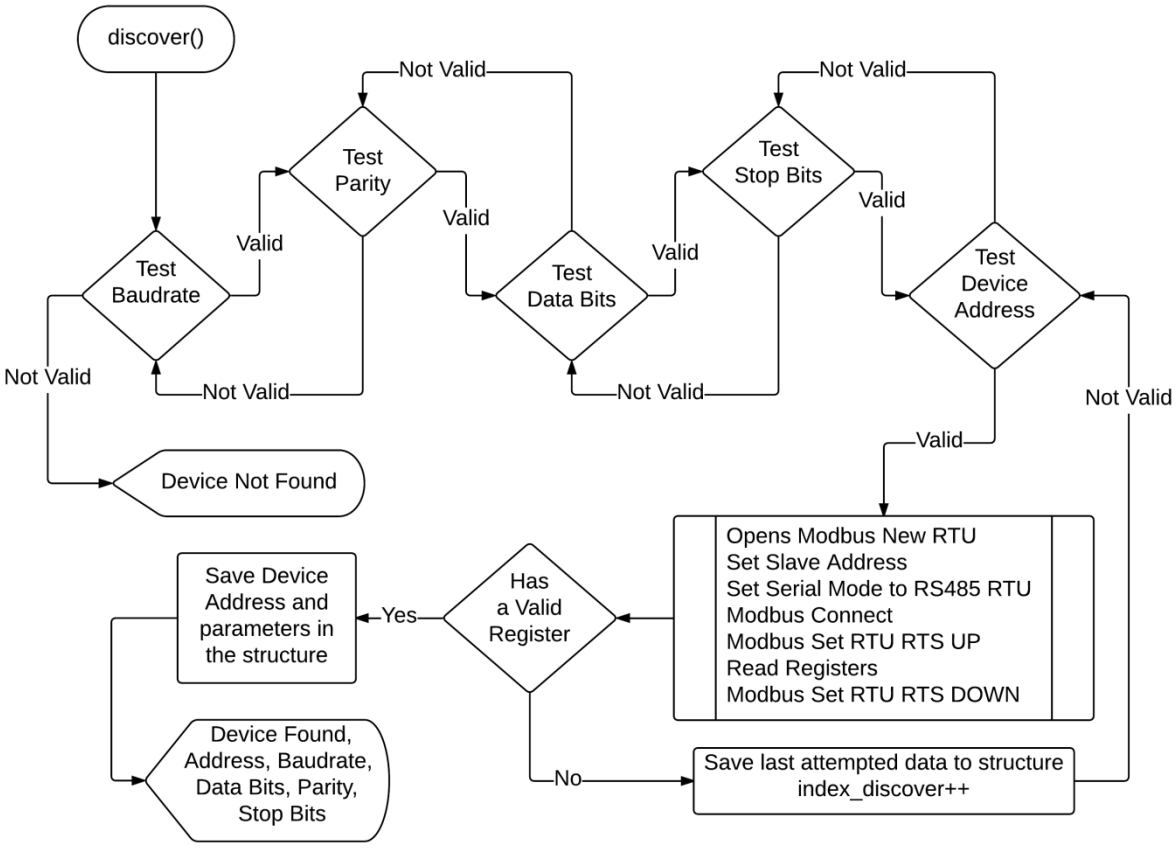

Figure 32 - *Modbus* Application "discover" function Flowchart

The `discover()` function flowchart, depicted in the Figure 32 is a “brute force” algorithm, that aims to find any device connected in the bus within the predefined parameters. The communication parameters are kept in vectors with the predefined values of the baud rate, parity, data bits, stop bits and device address. When the `discover()` function starts, it sets the first value for the baud rate, then it sets the parity bit, then the data bits, then the stop bits and finally sets the device address. With this first parameters set, it searches all the devices addresses within the predefined range of addresses. The parameters of each device found on the bus are kept in a structure. To detect a device in the bus, at least one register has to be successfully read. The function that reads a single register can also read a range of registers, however with the amount of possible registers that a single device can hold this process would consume too much time to be effective. When the read function ends it prints the devices found. This algorithm can find as many devices in a specific bus, since devices in the same bus can’t work with different communication parameters. In this implementation it’s assumed that the registers address are known and unlikely to be changed. This function can be preconfigured to search for several registers with the cost of more processing time. For this implementation, the address 0x07 was chosen, if the partner company, *TNL*, decides for instance to change the start address to 0x08, the device is no longer detected by this algorithm, unless the read register function is also changed. The `discover()` function was developed as a precaution measure. In case of the company *TNL* decides to change the communication parameters without prior information, this algorithm will be able to find the devices in the bus. The function `getData()` depicted in the Figure 33 is the flowchart of the function developed to read the registers from the *HUB*.

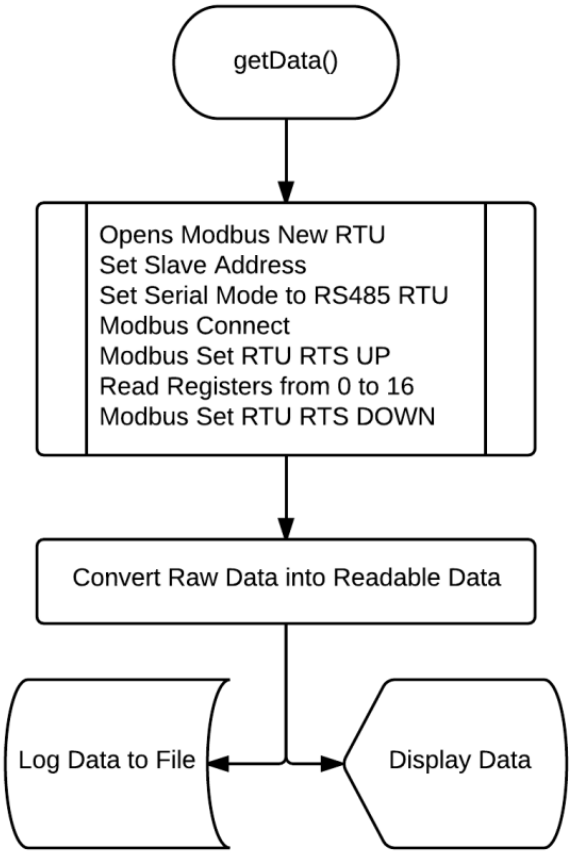

Figure 33 - *Modbus* Application getData() function Flowchart

The function `getData()` starts by a set of predefined procedures. It starts by opening a new "*Modbus* RTU" connection, sets the slave identity (ID) intended to retrieve data from, sets the serial mode communication to "RS485 RTU", starts the *Modbus* connection and set the RTS *UP* to transmit. The function to read the holding registers reads the registers from 0x00 to 0x10. After sending the request message it sets the RTS *DOWN* to listen the bus for a reply. The reply message carries the 17 registers with the data collected from the garbage container sensor. Those registers values are converted into readable data and logged into a file with a timestamp and displayed in the console. This data can be parsed through a script or interpreted by a user.

The Figure 34 depicts the request message format and the RTS signal sent from the Master to the slave at the physical layer. According the *Modbus* protocol the request message is sent to the Slave device with the address 0x40, the function code is the 0x03 (read holding registers function), the start address 0x00 (high and low), the number of registers to be read (high and low) 0x11 (17 registers from 0 to 16) and the last 2 registers are the Cyclic redundancy check (CRC).

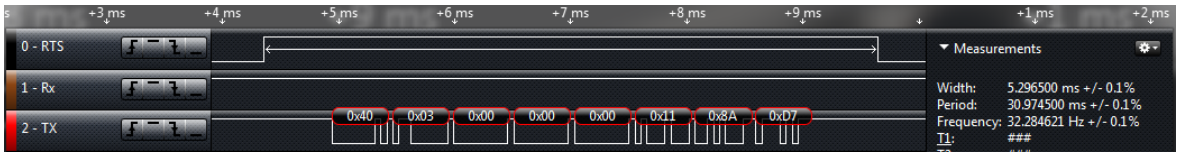

Figure 34 - *Modbus* Protocol Raw Request Message

The Figure 35 shows at the physical layer the message reply. The Master sets the RTS signal to LOW and listens for the message reply from Slave.

Figure 35 - *Modbus* Protocol Raw Reply Message

The Figure 36 shows at the physical layer level the raw message exchange (request and reply) between the Master (Raspberry Pi) and the Slave (*HUB*), the RTS signal *UP* while transmitting and *DOWN* while listening.

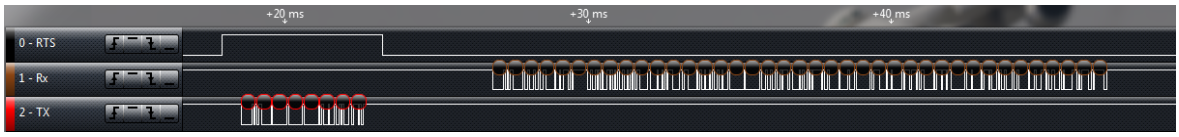

Figure 36 - *Modbus* Protocol Raw Message Request, Reply and RTS signal

When a device within the RS485 bus wants to transmit it has to set the RTS signal *UP*, and when intends to listen, it has to set the RTS signal *DOWN*, meaning that in the RS232 to RS485 converter this will be equivalent to the $\overline{RE}$ and DE pins have 3.3V when RTS signal is *UP* and 0V when RTS signal is *DOWN*. Listening to the control line is a technique used when there is more than one Master device on the RS485 bus to avoid collisions.

The Figure 37 shows the structure of the reply message in a raw format from Slave (*HUB*) with the values that each register holds. The *UUID* are identifiers, the *DIST* the amount of garbage inside the container in centimeters, the *ACC_X*, *ACC_Y* and *ACC_Z* are the 3 axis of the accelerometer, the *TEMP* is the temperature inside the container, the *POS_LAT_H*, *POS_LAT_L* and *POS_LON_H*, *POS_LON_L* are the Global Positioning System (GPS) coordinates (latitude and longitude), the *ACT* is the number of refuse collections and the *BAT* is the level of the battery voltage.

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
UUID0	UUID1	UUID2	UUID3	UUID4	UUID5	DIST	ACC_X

0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10
ACC_Y	ACC_Z	TEMP	POS_LAT_H	POS_LAT_L	POS_LON_H	POS_LON_L	ACT	BAT

Figure 37 - Message Format Garbage Street Containers

The Figure 38 illustrates the format of the reply message from the *HUB* after the conversion to decimal format data and where the GPS coordinates are converted into their original format.

SYST	UUID0	UUID1	UUID2	UUID3	UUID4	UUID5	DIST	ACCX	ACCY	ACCZ	TEMP	PLAT	PLON	ACT	BAT

Figure 38 - Message with readable data

To convert the GPS coordinates, the *POS_LAT_H* was collated with the *POS_LAT_L*, converted into a float type and divided by 1 million to recover the true value of the coordinate. The same process was applied in the longitude. The Figure 39 shows a print screen from the application developed and how it displays the data collected from the *HUB*. On the data output is added a timestamp and logged into a ".log" file.

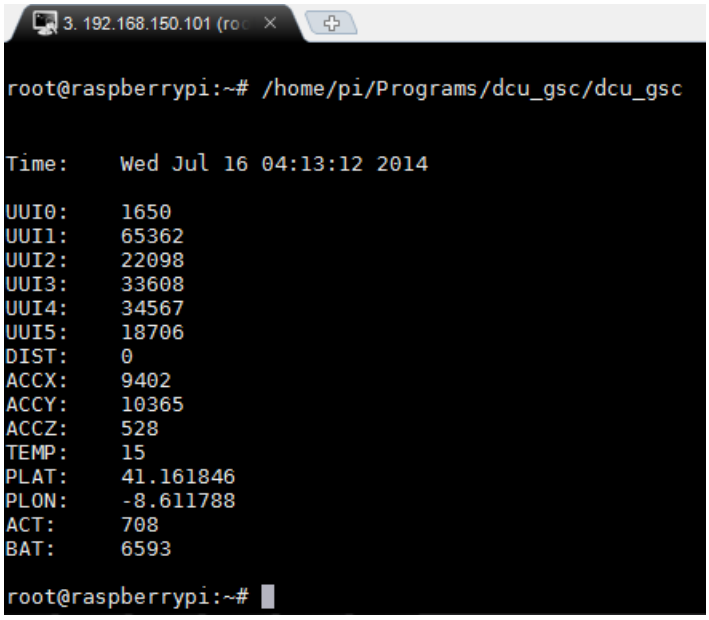

Figure 39 - Print Screen from DCU Application

The output depicted in the Figure 39 can be easily parsed by another application or interpreted by a user. The log file is where the information collected is stored in a persistent manner. This application was not developed to be used as a daemon, therefore is used upon request. The sensor reading time cycles round the 4h and can be changed by

TNL. Therefore with this time cycle, it only makes sense to run this application in a 4h time cycle.

5.4. PROPOSED MULTI-HOP NETWORK EXTENSION FOR DCU

Multi-hop networks are used within the scope of this dissertation in order to extend the existent network. The proposed architecture for this Multi-hop network extension is divided in 3 main items. The Data Collection units (DCUs), that collect data from garbage street containers, the Multi-hop network and the Urban Sense cloud database (DB). The DCUs collect the data from the garbage street containers and deliver it to the Multi-hop network. The Multi-hop network extends the *BusNet* in such way that remote DCUs from garbage containers that are not within the public Wi-Fi Access Points (APs) coverage neither within the *BusNet* range can send their data to the Urban Sense cloud DB. The proposed architecture of the Multi-hop network extension is represented in the Figure 40 by 3 nodes, however, this Multi-hop network can be composed by up to "N" nodes. While every node of the Multi-hop network is equipped with one Wi-Fi interface, the nearest node to the *BusNet* or a public AP is equipped with two interfaces. This allows the last node to act as a Multi-hop Network Gateway and send all the traffic generated from DCUs, within the coverage of the Multi-hop Network, to the Urban Sense cloud DB.

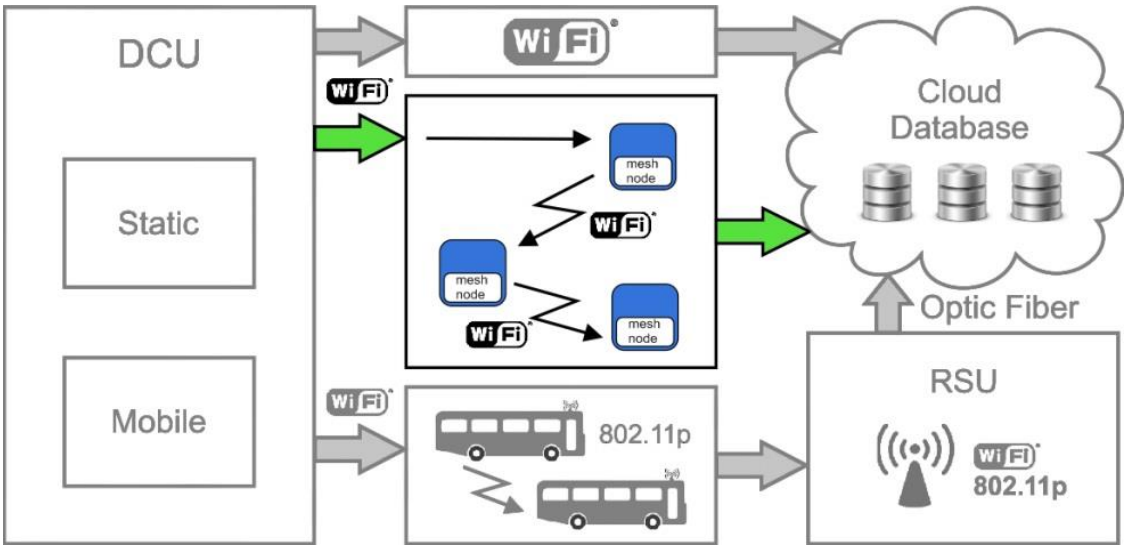

Figure 40 - Multi-Hop Network Extension Architecture

The hereby proposed architecture depicted in the Figure 40, aims to solve the problem depicted in the Figure 1 on the introduction chapter.

5.5. IMPLEMENTATION OF THE MULTI-HOP NETWORK EXTENSION

To implement the Multi-hop network extension presented in the previous section, two routing protocols were tested: Optimized Link State Routing Protocol (OLSR) and *Babel*. In both cases open source implementations were used and configured to run in Raspberry Pi. The OLSR implementation is available in [36]. The *Babel* implementation is available in [68]. The Ad-hoc On-Demand Distance Vector (AODV) routing protocol was a first choice but its viability ended right on the beginning of the implementation due to kernel legacy support from Raspberry Pi platform. To implement the Multi-hop Network three methods to configure the nodes were explored as well as other general configurations such as IP forwarding, iptables, Network Address Translation (NAT) and applications for Domain Name Service (DNS) such as "Avahi".

5.5.1. MULTI-HOP CONFIGURATION METHODS

The first method consists on the configuration of the interfaces file, located in the, `/etc/network/interfaces` of each node. The configuration files from relay nodes are almost equal, except for their unique assigned static Internet Protocol (IP) address. The interfaces and configuration files for the relay nodes are available in [69]. The Code excerpt 7 refers to the configuration of the first relay node.

```
# INTERFACE CONFIGURATION FOR OLSR
allow-hotplug wlan0
iface wlan0 inet static
address 192.168.83.101
netmask 255.255.255.0
wireless-essid mesh
wireless-mode ad-hoc
wireless-channel 6
wireless-power on

# INTERFACE CONFIGURATION FOR BABEL
allow-hotplug wlan0
iface wlan0 inet static
address 192.168.81.101
netmask 255.255.255.0
wireless-essid mesh
wireless-mode ad-hoc
wireless-channel 6
wireless-power on
```

Code excerpt 7 - First relay Multi-hop network node interfaces file configuration

The configuration done consists on the definition of the Wi-Fi interface parameters so it can boot in Ad-hoc mode and with a defined static IP address. After the Wi-Fi interfaces are configured and running, the routing protocol is called manually.

To call Optimized Link State Routing Protocol (OLSR) routing protocol the following command was issued:

```
$sudo olsrd -d 3 -i wlan0 &
```

The issued command runs OLSR protocol daemon with a debug level 3 (this may vary from 0 to 9, with 0 being to debug and 9 the most verbose). The level 3 shows the neighboring table and the metrics for each neighbor node. The interface that OLSR uses is defined to "wlan0" which is the interface configured previously as Independent Basic Service Set (IBSS). The "&" flag at the end is to attribute a new Process Identifier (PID) to the process run in background.

To call *Babel* routing protocol the following command was issued:

```
$sudo babeld -D -d 3 -z 3 -c /etc/babeld.conf
-S /var/lib/babeld/state wlan0 &
```

The issued command runs *Babel* protocol as a daemon with a default debug level 3 dumping all the interaction with the Operating System (OS) kernel. The debug level varies between 0 and 3, according the verbosity intended. The flag "-z" enables diversity-sensitive routing. Being 0 (no diversity), 1 (per-interface diversity with no memory), 2 (per-channel diversity with no memory), or 3 (per-channel diversity with memory). The value factor specifies by how much the cost of non-interfering routes is multiplied, in units of (1/256). The "-c" flag specifies the configuration file. The "-S" flag sets the name of the file used for preserving long-term information between invocations of the "babeld" daemon.

The configurations for the Multi-hop gateway node differ from relay nodes since this node has two interfaces, one configured as a station (infrastructure mode) connected to an Access Point (AP) and the other configured in Ad-hoc mode. The Code excerpt 8 refers to the configuration of the interface in infrastructure mode in the gateway node.

```
# INTERFACE CONFIGURATION FOR STA wlan0
allow-hotplug wlan0
iface wlan0 inet dhcp
wireless-essid WiFi Porto Digital
```

Code excerpt 8 - Multi-hop network gateway node interfaces file configuration

The configuration done consists on the definition of the Wi-Fi interfaces parameters so one interface boots in managed mode with IP assigned by Dynamic Host Configuration Protocol (DHCP). The other interface is configured in Ad-hoc mode, like in the relay

nodes and boots in Ad-hoc mode with a defined static IP address. After the Wi-Fi interfaces are configured and running, the routing protocol is called the same way is called in relay nodes. The interface “wlan1” was the selected interface to run OLSR and *Babel* in the gateway node. For this first method the configuration files for each node and corresponding protocol are available in the Appendix G, Appendix H and Appendix I.

The second method consists on turning the first method automatic. The interface configuration is done the same way as in the first method. The way the routing protocols now start, has become automatic. The OLSR routing protocol provides an option that allows the user to define whether the OLSR routing protocol starts on boot or not. This option can be defined in the file `/etc/default/olsrd`:

```
START_OLSRD="YES"
```

By defining “YES”, the OLSR daemon starts on boot.

The *Babel* protocol upon installation does not provide any configuration file. To set the *Babel* routing protocol to start on boot the file `/etc/rc.local` was edited and the following line added:

```
$/etc/init.d/babeld start
```

The `/etc/rc.local` file is a script that runs every time the Raspberry pi boots, therefore the *Babel* routing protocol can start running with all interface configurations on boot.

The third method consists on making the Multi-hop network nodes configuration fully automatic. This is the most suitable solution for this dissertation work, therefore the one used. For this purpose, four scripts were developed. To start OLSR routing protocol the `olsr.start` is called, to stop OLSR is called `olsr.stop`. To start the *Babel* routing protocol the `babel.start` is called, to stop *Babel* is called `babel.stop`. Each Multi-hop network node has both configuration scripts. The relay nodes configuration is almost equal except for their unique static IP address. The node101 script for OLSR and *Babel* is presented in the Appendix J. The node102 script for OLSR and *Babel* is presented in the Appendix K. The node103 acts as the gateway in the Multi-hop network, therefore its script is a bit different from the relay nodes script. The node103 script for OLSR and *Babel* is presented in the Appendix L. The scripts with extension (.stop) will stop the routing protocol deamon and set interface down. These scripts are loaded into the file `/etc/rc.local`, according the intended protocol to run on the Multi-hop network. The

developed scripts define the Wi-Fi interface and set it up and running, along with the intended routing protocol. No configurations are done in the `/etc/network/interfaces` and regardless of what is defined in it, this method will override it.

The Multi-hop network nodes configuration is a major part of the network, however for the multi-hop network to work correctly the nodes of the network need to have the capability to forward traffic, therefore the IP Forwarding had to be activated. To activate the IP forwarding capability, the following command was issued:

```
$sudo echo 1 > /proc/sys/net/ipv4/ip_forward
```

To make this change permanently the following file had to be edited with the command:

```
$nano /etc/sysctl.conf
```

Inside the file, the following line was uncommented:

```
#net.ipv4.ip_forward=1
```

The IP Forwarding is now active.

The *iptables* is an application that allows the configuration of the Linux kernel firewall. The *iptables* applies to IPv4, *ip6tables* to IPv6, arptables and to Address Resolution Protocol (ARP). To implement the correct iptables rules two scripts were developed. The script `iptables.start` and the script `iptables.stop`. Both scripts are present in every node of the multi-hop network. The `iptables.start` script from Code excerpt 9 is only present on the gateway node.

```
#!/bin/sh
#This script has all the rules in the iptables.
echo "Flushing all the rules in the iptables..."
iptables -F
iptables -t nat -F
iptables -t mangle -F
iptables -X
# Rules
echo "Running all the rules into the iptables..."
#Allow Internal Network to External Network Access and Vice Versa
iptables -A FORWARD -i wlan1 -o wlan0 -j ACCEPT
iptables -A FORWARD -i wlan0 -o wlan1 -j ACCEPT
# Enable NAT
sudo iptables -t nat -A POSTROUTING -o wlan0 -j MASQUERADE
# Enable IP Forward.
sudo sh -c echo 1 > /proc/sys/net/ipv4/ip_forward
# Saving Rules
iptables-save > /etc/network/iptables.rules
echo "Done!"
exit 0
```

Code excerpt 9 - Iptables script for Multi-hop network gateway node

The rules applied in the iptables refer to the gateway node (node103) of the Multi-hop network, hence the need to perform NAT. The script developed for iptables includes a IP forward for redundance purposes. At the end of the script the rules are saved into a file called `iptables.rules`. This file is later used to load the configurations done in `iptables.start`, to iptables upon boot.

To turn iptables rules persistent the script "`iptables`" was developed as the Code excerpt 10 shows and placed in the `/etc/network/if-pre-up.d/iptables`.

```
#!/bin/sh
#This script restores iptables upon reboot
iptables-restore < /etc/network/iptables.rules
exit 0
```

Code excerpt 10 - Iptables script to restore rules on boot

This way everytime the Raspberry Pi boots up, the iptables rules are loaded. This applies to every node of the multi-hop network.

The script `iptables.start`, represented in the Code excerpt 11 is the script for the relay nodes.

```
#!/bin/sh
#This script has all the rules in the iptables.
echo "Flushing all the rules in the iptables..."
iptables -F
iptables -t nat -F
iptables -t mangle -F
iptables -X
# Rules
echo "Running all the rules into the iptables..."
#Allow Traffic Forwarding
iptables -A FORWARD -i wlan0 -o wlan0 -j ACCEPT
# Enable IP Forward.
sudo sh -c echo 1 > /proc/sys/net/ipv4/ip_forward
# Saving Rules
iptables-save > /etc/network/iptables.rules
echo "Done!"
exit 0
```

Code excerpt 11 - Iptables script for Multi-hop network relay nodes

The script `iptables.start` ensures that the Linux firewall is not blocking any traffic being forwarded and activates the IP forwarding for redundance purposes.

The script `iptables.stop` represented in the Code excerpt 12 is common for every node of the Multi-hop network. This script just flushes all the tables existent in the iptables and is as follows:

```
#!/bin/sh
#This script clean all the rules in all the tables of the iptables
echo "Stopping firewall and allowing everyone..."
iptables -F
iptables -X
iptables -t nat -F
iptables -t nat -X
iptables -t mangle -F
iptables -t mangle -X
iptables -P INPUT ACCEPT
iptables -P FORWARD ACCEPT
iptables -P OUTPUT ACCEPT
echo "Done!"
exit 0
```

Code excerpt 12 - Iptables script for flush all rules

The “Avahi” is a free zero-configuration networking (zeroconf) implementation, including a system for multicast DNS/DNS-SD service discovery. “Avahi” is an application which enables programs to publish and discover services and hosts running on a local network. To install “Avahi” the following packages were installed: `libnss-mdns avahi-utils`. This installation requires the kernel module IPv6 loaded. Although neither of the protocols used in this work was used with its IPv6 address, they do have the capability to route IPv6. The *Babel* protocol for instance, demands that the kernel module IPv6 has to be loaded for proper working.

5.5.2. *PINGLED* SCRIPT FOR FIELD DEPLOYMENT

The *pingLED* is a script used on every Multi-hop Network nodes and it’s used for field deployment. The `pingLED.sh` is a crafty script developed to allow visual support for field tests. With this script the user can acknowledge if the node is working correctly in the network upon boot without the need of an extra accessory. This script is implemented in every node of the Multi-hop network and uses the *ACT* Light Emitting Diode (LED) built on board of the Raspberry Pi according the Figure 41. This acknowledges the user if this node is connected to the network correctly.

Figure 41 - Raspberry Pi ACT LED

In every node of the network this script starts on boot and starts to ping the Multi-hop network gateway. The gateway of the network has also this script, however in this particular node it pings an external entity to ensure that external network is accessible. As depicted in the Figure 42 upon start, this script disables the normal operation of the *ACT* LED. Then the number of attempts is loaded with a maximum value and starts to ping the gateway, if it returns a valid ping, the Raspberry Pi will turn the *ACT* LED "ON" and set the number of attempts to zero. If the ping fails, it starts incrementing the number of attempts until it gets a valid ping and set it to zero again, or until it reaches the maximum value defined and closes the script. Before the script closes, it sets the *ACT* LED to its default operation mode. While in operation, this script monitors any kind of signal that comes from inner Raspberry Pi working operation and monitors the `iperf` process. If some of the signals defined in the script is trapped, the script terminates. If the `iperf` process starts, this script will detect it and closes itself. This was made for testing purposes in order to avoid data flowing in the network unless the intended one. This script was built to be crash proof. If something goes wrong while this script is running, this script will reset the *ACT* LED to its default operation mode.

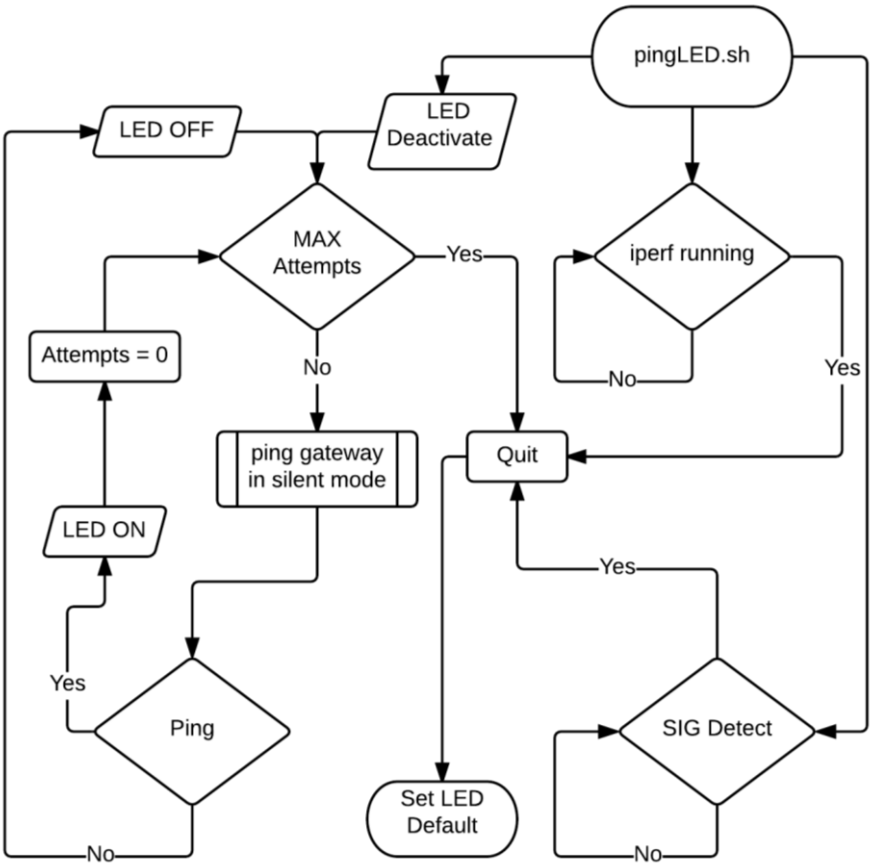

Figure 42 - *pingLED* application flowchart

The *pingLED* script is very handy on the field. It provides real time visual information to the user of when and where the Raspberry Pi is within the network range. In a Multi-hop scenario it works very well, because upon deployment of the nodes it's possible to instantly have visual information whether it's in range or not and if not in range, the user just needs to follow the hops, until the node that has the LED "ON". This means the previous node can't reach the gateway through the Multi-hop network. The *pingLED* script was tested on the field and proved to be up to the task. This script is available in the Appendix C.

5.5.3. MULTI-HOP IMPLEMENTATION SETBACKS

The setbacks on the implementation of this Multi-hop network started in the first attempt to use the gateway node with a single Wi-Fi interface to connect between two different network topologies. A solution was to use Wi-Fi interface virtualization to connect the same physical device to two distinct modes simultaneously. However the Wi-Fi interface driver does not support virtualization of the same physical device into two distinct modes simultaneously station (STA) and Independent Basic Service Set (IBSS). The solution was to use two physical devices, one to work in STA mode, in the infra structured network and the other in IBSS mode, to connect to other IBSS or Ad-hoc peers in the Multi-hop network.

Installing the *Babel* routing protocol is a straightforward task. However its configuration demands a deep knowledge of the protocol. The first problem on implementing *Babel* in Raspberry Pi is the inexistence of its configuration file after *Babel* installation. The configuration file allows *Babel* daemon "babeld" to get the kernel routes and disseminate them across the Multi-hop network. As the author of *Babel*, Juliusz Chroboczek states in [70], "This is experimental software, run at your own risk.". He also states that the rejected routes issue can be overcome through "–S" or "–r" flag option to use the persistent file from node identity (ID), however the use of any of these flags seem to make no effect. When Multi-Hop network nodes start with *Babel* routing protocol they get rejected routes, this is shown in their own routing table by the "!H" flag (rejected route). To overcome this situation, in the script `babel.start`, the "babeld" is started, waits for 1 second and stops "babeld", repeats this process for one more time and finally starts the "babeld". Although it's stated in [70] that after 4 minutes (default), the nodes should recover routes, such thing didn't happened even with longer periods of up to 12 hours. This is a major problem

because nodes either upon boot or started through `babel.start` script can fall in this situation and get isolated with no Wi-Fi access, leaving the option of: use Secure Shell (SSH) through the "eth0" interface or in last resort, perform a hard reset (unplug the power supply and connect again), since the Raspberry Pi platform has no reset button.

5.6. SCRIPTS FOR TESTING PURPOSES

The purpose of the testing scripts developed within the context of this dissertation work emerged with the need to start specific applications on boot with predefined options as well as log the results outputted by them. The testing purpose scripts were used to test and debug the work developed in the several stages of its implementation.

5.6.1. SHARK SCRIPT

The `shark.sh` script was developed to start the "tshark" application with predefined options. This provides an easy and faster use of "tshark" application within the scope of the tests performed on this dissertation work. The `shark.sh` was written in bash and starts on boot with predefined options. This script is used to log the packets exchanged right on the beginning of the Multi-hop network formation. In case of no arguments are defined it starts using the Wi-Fi interface "wlan0" with a duration of 5 minutes, saving all the traffic exchanged within that period in a (.pcap) file named "tshark" with date and timestamp.

The `shark.sh` script can also be started manually and arguments can be provided. The "help", "time" and "packets" are the first position valid arguments. The "help" provides a list of interfaces (second argument), where "tshark" can listen, as well as the instructions on how to use it. The "time" argument defines the third argument as the amount of time the script is running before it stops. The "packets" argument defines the third argument as the amount of packets that the script counts until it stops. The source code of this script is available in the Appendix E.

5.6.2. SPEED TEST SCRIPT

The Speed Test Script is a python script developed by Matt Martz [71] and is used in the context of this dissertation work to measure the uplink and downlink from the Multi-hop network nodes. To take measurements and log them, two simple scripts were developed:

The `speedtest.sh` represented by the Code excerpt 13:

```
#!/bin/bash
OUTPUT=`./speedtest-cli`
echo -e "$OUTPUT\n$RIGHTNOW\n############\n"
>> st_results
```

Code excerpt 13 - Speed Test

The `speedtest_simple.sh` represented by the Code excerpt 14:

```
#!/bin/bash
OUTPUT=`./speedtest-cli --simple`
echo -e "$OUTPUT\n$RIGHTNOW\n############\n"
>> st_results_simple
```

Code excerpt 14 - Speed Test Simple

Both scripts start the application `speedtest-cli`, however the options differ. The `speedtest-simple.sh` represented by the Code excerpt 14 retrieves the ping latency and the speeds from upload and download. The `speedtest.sh` represented by the Code excerpt 13 retrieves the previous information and adds more detail, such as the server that was used to perform the test (based on latency) and the distance to it.

5.7. ON BOARD UNIT APPLICATION

The On Board Unit (OBU) depicted in the Figure 43, it's a proprietary device from *VeniamWorks* also referred as NetRider. The OBU is installed inside the buses, trucks and taxis that form the Vehicular network. Essentially the OBU works as a router. It's equipped with a Microprocessor without interlocked pipeline stages (MIPS) processor and it's composed by several interfaces, such as: an antenna for IEEE 802.11p - Dedicated Short-Range Communications (DSRC) (5.9 GHz) to communicate with the vehicular network, an antenna for communication with Wi-Fi (2.4 GHz) acting as a station (STA) in an infra structured network, Basic Service Set (BSS), an antenna for GPRS/3G/4G, a Global Positioning System (GPS) antenna, an ethernet interface, a serial communication port, a 3 axis accelerometer, a 3G mobile card and a Bluetooth device.

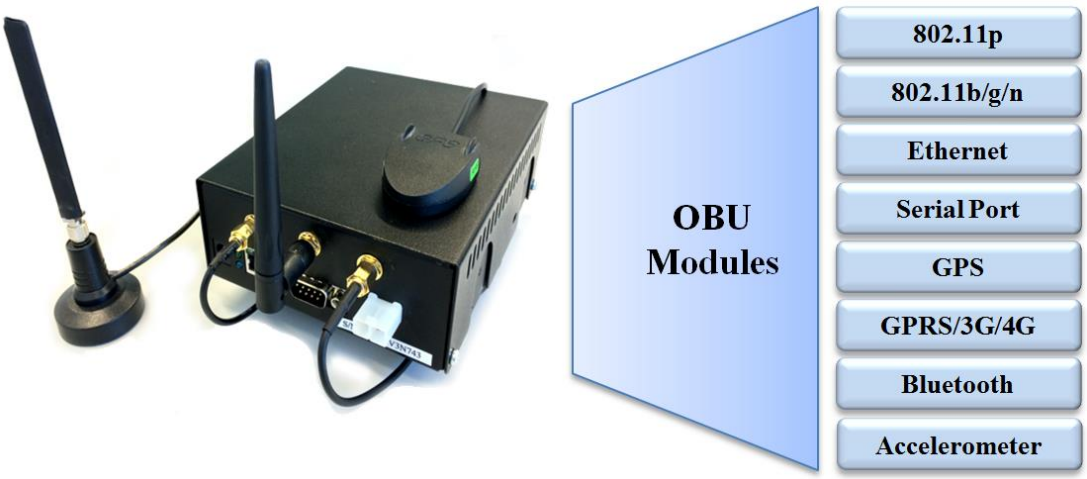

Figure 43 - On Board Unit (OBU)

This unit provides vehicles a wide range of communications, such as Vehicle-to-vehicle (V2V), Vehicle-to-roadside (V2R) and Vehicle-to-infrastructure (V2I). The Operating System (OS) running inside these OBUs is a proprietary customized compilation of the open source *OpenWrt* [72] OS. The OBUs have a connection management software that select the most convenient communication protocol to be used, according the situation. The OBUs use the vehicles as data mules.

The OBU in the context of this dissertation work is used to get real time global position of each vehicle, its speed, if it's stopped or in idle and its acceleration in the 3 axis. To provide this functionality, a Transmission Control Protocol (TCP) Server was implemented. Because there isn't enough memory/disk space on the OBU to install a compiler and compile the code natively, it's needed to cross-compile the code on a development environment to a target system. Prior knowledge from *OpenWrt* [72] OS had to be acquired.

5.7.1. BUILD ROOT FOR MIPS CROSS-COMPILE

Embedded systems like the one used in the On Board Unit (OBU) use a different processor and require a cross-compilation toolchain. A compilation toolchain runs on a host system but generates code for a target system. The cross-compilation toolchain uses *uClibc*, a tiny "C" standard library. In this dissertation work the host system uses a virtualized x64 architecture and distribution of Linux, Debian7 Operating System (OS) kernel version 3.2.60-1+. The target system, the OBU, uses a Microprocessor without interlocked pipeline stages (MIPS) architecture. To cross-compile, a special compiler and development

environment called the *OpenWRT* Software Development Kit (SDK) is needed. The SDK can be obtained in [73]. The SDK used for this purpose was the "OpenWrt-SDK-Linux-i686-1", however it varies depending on the architecture were the application is being developed, the architecture of the OBU and the version/release of *OpenWrt* the OBU is running. *OpenWrt* Buildroot is a set of Makefiles and patches that allow users to easily generate both a cross-compilation toolchain and a root file system for embedded systems. To configure the kernel the configuration file `config_aveiro` was provided by VeniamWorks to replicate the kernel in the OBU. After the kernel compilation has finished the cross-compilation was ready to be used to compile applications for the OBU. To cross-compile the TCP Server Application a Makefile, available in Appendix A, was developed.

5.7.2. TCP SERVER APPLICATION

The Transmission Control Protocol (TCP) Server application developed for the On Board Unit (OBU), allows users and/or applications acquire in real time information of where and when a specific vehicle is located. The TCP Server application is composed by three files, the `tcpServerMIPS.c`, the `gps.c` and the `accelerometer.c`, according the Figure 44.

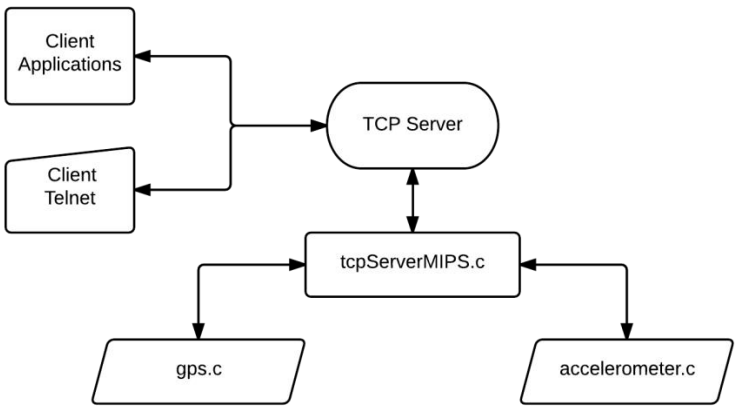

Figure 44 - OBU TCP Server Application Structure Diagram

The file `tcpServerMIPS.c` is the main file and calls the functions in `accelerometer.c` and `gps.c`. After compilation, this application becomes a single executable file.

To start the application on the OBU, the command: `./tcpServerMIPS` is issued on the console. A client connects to the OBU Internet Protocol (IP) address and default port: 4444. To run the *telnet* application in a client, the command: `telnet <IP> <PORT>` is issued. The Figure 45 is a flowchart from the TCP Server Application.

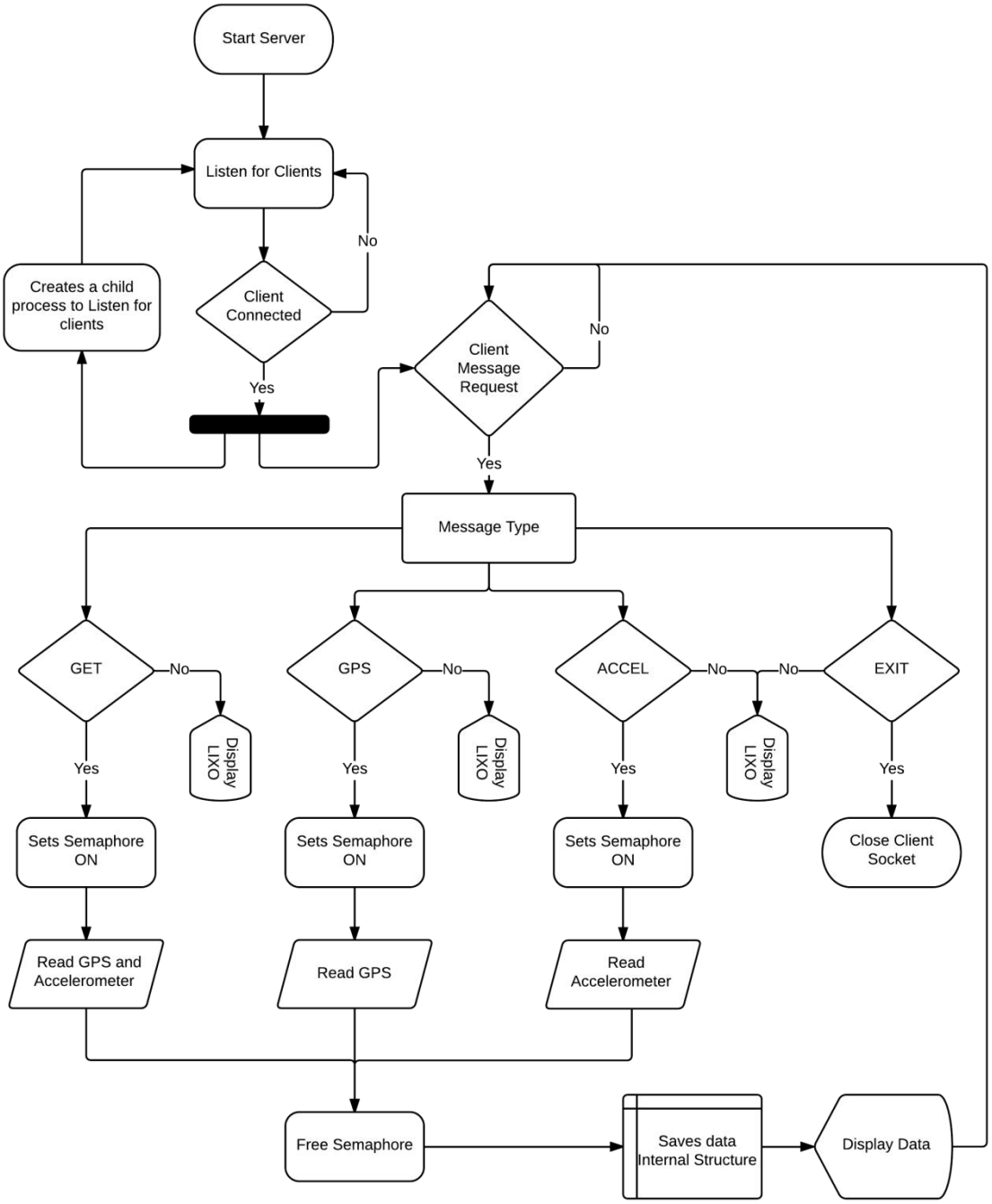

Figure 45 - TCP Server OBU Application Flowchart

The TCP Server application available in the Appendix F was developed to hold up to 5 clients simultaneously and starts by listening for clients. There are 4 types of messages that can be exchanged between the TCP Server Application and the clients. When a client sends a request message, i.e. "GPS", the server application calls the function `gps()`. This function accesses the Global Positioning System (GPS) and retrieves the GPS data according the Figure 46. During this process if any error occurs, the GPS function creates

an error log file. The `gps()` function has also the capability to create a file and save all data in a file instead of only printing it, however by default this functionality is not active to save memory space. When a client sends a request message i.e. "ACCEL", the server application calls the function `accelerometer()`. This function works with the same operation principle of the `gps()` function, however this function deals with the 3 axis of the accelerometer and the status information of the vehicle. When a client sends a request message i.e. "GET", the server application calls both functions, the `gps()` and the `accelerometer()`. When a client sends a request message i.e. "EXIT", the server application closes the connection for the client and waits for other clients to serve. Other messages are not considered, therefore if the server receives from any client a message different from the previous referred, the server replies with the message "LIXO".

The TCP Server Application sends the information between the "INI" and "END" identifiers to delimit the messages. This format allows a better perception for users and an easy way to parse information from applications. In the Figure 46 is depicted the "GPS" message format.

INI	SYST	GPST	GFIX	NSAT	HDOP	GLAT	GLON	GALT	SPED	HEAD	STAT	END

Figure 46 - TCP Server GPS Format Message

The "GPS" message payload is composed by 11 variables: the system time (SYST), the GPS time (GPST), the fix (GFIX) used for accuracy, the number of sattelites (NSAT), the horizontal dilution of precision (HDOP) used for accuracy, the latitude (GLAT), the longitude (GLON), the altitude (GALT), the speed (SPED), the direction heading in degrees to north (HEAD) and the status of the vehicle (STAT). In the Figure 47 is depicted the "ACCEL" message format.

INI	SYST	GPST	ACCX	ACCY	ACCZ	STAT	SPED	END

Figure 47 - TCP Server Accelerometer Format Message

The "ACCEL" message payload is composed by 7 variables: the system time (SYST), the GPS time (GPST), the acceleration in the *X* axis (ACCX), the acceleration in the *Y* axis (ACCY), the acceleration in the *Z* axis (ACCZ), the status of the vehicle (STAT) and the speed (SPED). The "STAT" is computed with the accelerometer data and is an application developed by *VeniamWorks*. In this dissertation work it's used to provide information of when the vehicle is stopped, moving, on idle and even its slope.

The Figure 48 shows a print screen from a *telnet* client connected to the TCP Server application, requesting a "GPS" message and retrieving the reply message. The reply message is obtained according the defined message format referred in Figure 46.

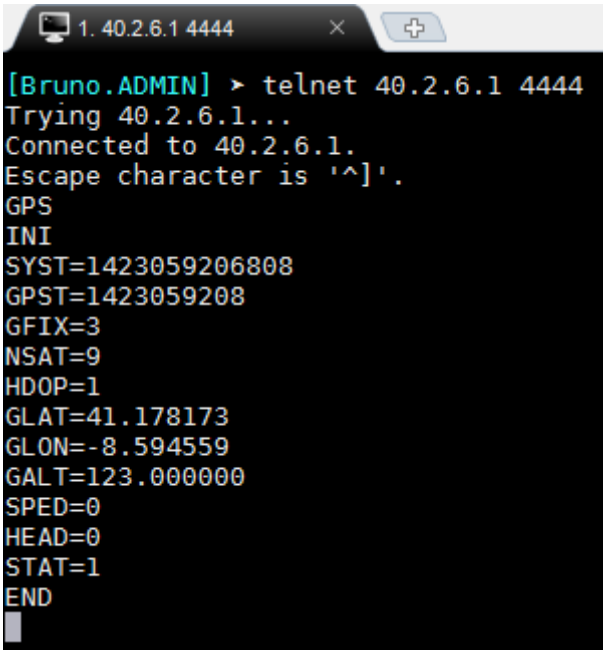

Figure 48 - OBU telnet client GPS message request and reply

Available in the Appendix M are all the exchanged messages between the TCP Server application and the *telnet* client.

5.8. SUMMARY

This chapter presents the proposed architecture and implementation realized in this dissertation work. It starts by describing the existent network architecture with the proposed Multi-hop network extension embedded. The proposed Multi-hop network extension aims to connect remote street garbage containers as well as provide a redundant path for the DCUs that are already covered by the *BusNet*, RSUs and/or APs. This chapter is divided in seven sections. The first section shows a table with Single Board Computers specifications that could be an alternative to the Raspberry Pi used in the Future Cities Project. The second section describes the architecture of the DCU for garbage street containers which is divided into a hardware architecture and a software architecture. The third section describes the implementation of the DCU for garbage street containers. It shows how the *HUB* from *TNL* connects to the Raspberry Pi at the physical level using a *RS485* to a *RS232* converter and how they comunicate using the "*Modbus* RTU" protocol

implementation. It's referred what had to be changed in the "*Modbus* RTU" protocol in order to meet the requirements from the *TNL HUB*. The application developed for the DCU for the garbage street containers is detailed and the messages at the physical level are shown with the handshake signals working properly. The messages at the application layer are also shown and described. The fourth section describes the proposed Multi-hop network extension for DCU highligting the work developed in this dissertation. The fifth section describes the implementation of the Multi-hop network extension by describing the three methods to configure the Multi-hop network nodes. The *pingLED* script is detailed and how this tool was used to deploy the Multi-hop network nodes on the field as well as is referred the Multi-hop implementation setbacks felt especially on the *Babel* routing protocol configuration. The sixth section shows the scripts developed with the purpose of testing the Multi-hop network. The seventh and last section describes the On Board Unit application developed to provide the vehicle GPS location as well as information of when the vehicle is stopped, moving, on idle or even its slope. It details the messages exchanged and their format. It also explains the limitations of the OBU and the need to Cross-compile.

6. FIELD EXPERIMENTS ON MULTI-HOP NETWORKS

In the field experiments performed on Multi-hop networks, two experimentations were conducted: (1) the Experimental Coverage Setup and (2) the Multi-hop Network performance setup. The first section is where the setup conditions and location where the Multi-hop deployment took place for the coverage setup are described. The Multi-hop concept is proven by showing an active route connection. The second section described the Multi-hop network performance setup and discussed the results obtained from the routing protocols Optimized Link State Routing Protocol (OLSR) and *Babel*. In both experiments the wireless interfaces used in all nodes, are from *TP-Link*, model *TL-WN722N*. The wireless transmit power is $20\ dBm$ and the receive sensitivity is $-90\ dBm$ @ $8\%\ PER$, according to the manufacturer [74]. These interfaces are equipped with an external antenna with a $4\ dBi$ gain, leading to an Effective Isotropic Radiated Power (EIRP) of $-6\ dB$. The channel 6 (2,437 GHz) was used because it's a non-overlapping channel in the 2.4 GHz Industrial Scientific Medical (ISM) band. In this dissertation work the OLSR and *Babel* where chosen as viable protocols. The OLSR protocol is used due to its maturity and longtime existence as an Ad-hoc protocol. Tests from many authors and its implementation on commercial products in real environments show its credibility [75][76][77][78][79]. The *Babel* protocol is used due to its hybrid characteristics. Both protocols are also

available in the latest Rasberry Pi distribution of Debian Wheezy for Advanced RISC Machines (ARM).

6.1. EXPERIMENTAL MULTI-HOP COVERAGE SETUP

The experimental coverage setup took place in an urban scenario, in the city of Porto, as depicted in the Figure 49. The design and deployment of this network seeks to obtain field measurements of the Wi-Fi coverage in an urban scenario. Depicted in the Figure 49, by a blue line is the physical path where Multi-Hop network nodes were deployed. The path starts at Rua Damião de Gois in Porto City (where the garbage containers are deployed) and ends at Praça do Marquês, where the Access Point (AP) is connected to the *Porto Digital* AP. This AP acts as a gateway for Urban Sense cloud database (DB) access. Represented by blue squares are the Multi-hop network nodes. The distance between each other, from left to right, are: the 1st node is 280m from 2nd node, the 2nd node is 280m from the 3rd node and the 3rd is 148m from the *Porto Digital* AP.

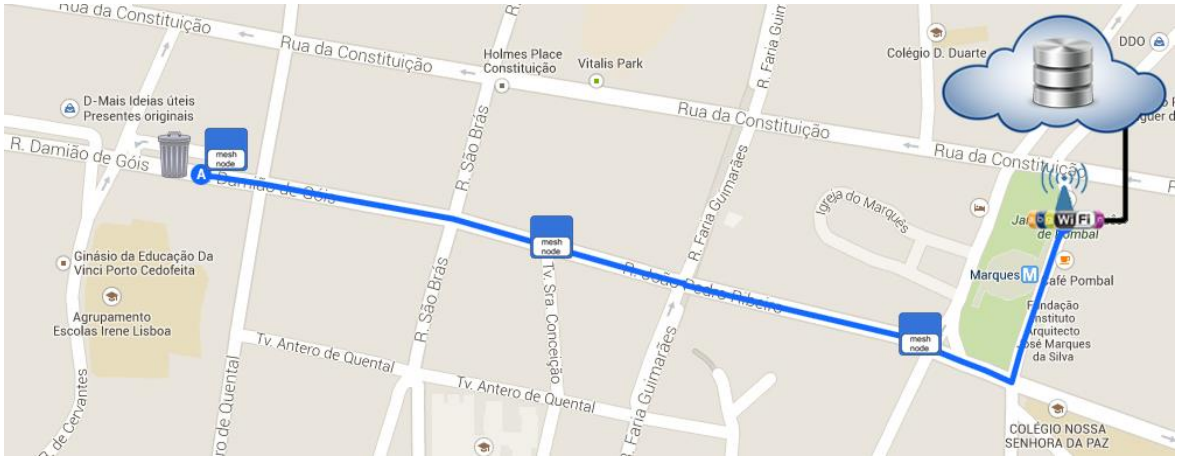

Figure 49 - Experimental Multi-hop coverage setup

The nodes distances were predicted by using the Radio Propagation Models as referred in the chapter 3 p.37, taking into account the Wi-Fi interface manufacturer features and the urban environment where the setup takes place.

To implement this setup at least 4 batteries were needed, one for each Multi-hop network node and one for the Wi-Fi Planner Tool. At the time that this setup was implemented, there was only one battery available. Therefore to accomplish the coverage survey on the field, a car with a Multi-hop network node was parked in the place of each node. The

Figure 50 depicts the car with the Multi-hop network node in each node position. To simulate the height of a semaphore (approximately 2m) the antenna was elevated.

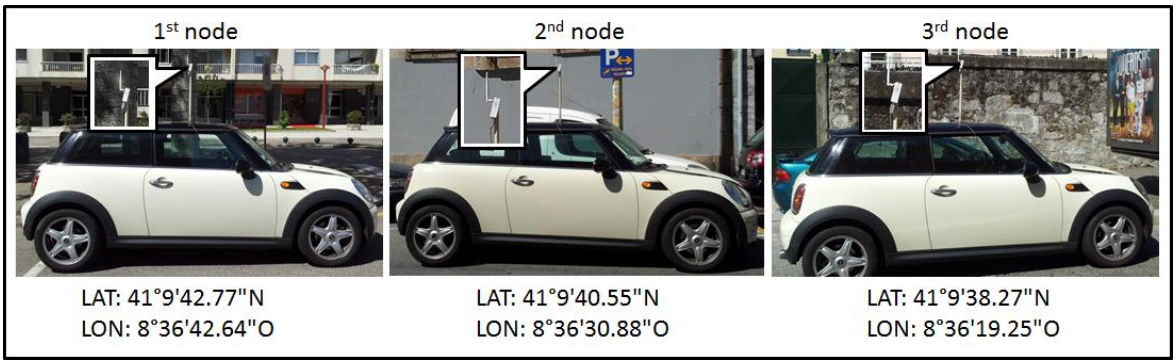

Figure 50 - Multi-hop coverage setup node location

The car was used to supply the power needed by the Multihop network node to work. The battery was used on the Wi-Fi Planner Tool. To start the measurements for the coverage survey the car was parked on the same place of the first Multi-hop network node. The Wi-Fi Planner Tool was moved throughout the whole route capturing the Service Set Identifier (SSID): "mesh" broadcasted by the Multi-hop network node. The car was then moved to the place of the 2nd node and the procedure was repeated until all the measurements were taken. This alternative to surpass the lack of batteries was possible because all the nodes of this Multi-hop network have the same characteristics and the only thing to be measured was the coverage survey.

6.1.1. COVERAGE SURVEY

In order to implement this Multi-hop network extension setup in the field, it was necessary to estimate the best position for each node of the Multi-Hop network and how many nodes were needed to get and end-to-end connection. To predict the optimized nodes position, the Radio Propagation Models described in the section 3.5, p.37 were used. For the Simplified Path Loss model a field measurement took place in order to optimize the value of the path loss exponent γ. The Table 11 shows the field measurements. The Table 12 shows the Path Loss obtained through the received power in the Table 11 and the transmitted power ($20\ dBm$).

Table 11 - Field Measurements

Distance (m)	Power Received (dBm)
10 m	-48 dBm
50 m	-64 dBm
100 m	-66 dBm
150 m	-61 dBm
200 m	-72 dBm
250 m	-79 dBm
300 m	-78 dBm
350 m	-76 dBm

Table 12 - Path Loss from Field Measurements with 20dBm Transmitted Power

Distance (m)	Path Loss (dB)
10 m	68 dB
50 m	84 dB
100 m	86 dB
150 m	81 dB
200 m	92 dB
250 m	99 dB
300 m	98 dB
350 m	96 dB

To estimate the nodes position, the value used for reference distance was $d_0 = 10m$. This value was chosen as this was the first sample to be taken and both the receiver and transmitter were in Line Of Sight (LOS).

The value of the path loss exponent γ was obtained through the minimum average quadratic error value. This was done using the equation [51, p. 39]:

$$F(\gamma) = \sum_{i=1}^{N_{samples}} [M_{measured}(d_i) - M_{model}(d_i)]^2$$

Where the $M_{measured}$ is the Path Loss (PL) in the Table 12 and the M_{model} was obtained through the equation [51, p. 39]:

$$M_{model}(d_i) = K - 10 \cdot \gamma \cdot log_{10}\left(\frac{d_i}{d_0}\right)$$

Where the K value is the PL obtained through the equation of Free-Space propagation model in relation to the reference point d_0. To find in the equation where is the minimal error, a derivative is needed and equate it to zero [51, p. 40]:

$$\frac{\partial F(\gamma)}{\partial \gamma} = 0$$

These steps return the optimized value for the path loss exponent γ where can be replaced in the Simplified Path Loss model to get accurate prediction for distance between transmitters and receivers in this specific environment. To predict the location and distance between transmitters and receivers the following calculations were made:

The first thing done after the measurements in the field were taken, was to obtain the optimized value of the path loss exponent γ. The value of $d = d_0 = 10m$ and the channel used was the channel 6 (2,437 GHz), therefore:

$$P_{r_{dBm}} = P_{t_{dBm}} + G_{t_{dBi}} + G_{r_{dBi}} + 20 \cdot log_{10}\left(\frac{\lambda}{4 \cdot \pi \cdot d_0}\right)$$

$$P_{r_{dBm}} - P_{t_{dBm}} = 4 + 4 + 20 \cdot log_{10}\left(\frac{\frac{3 \cdot 10^8}{2437 \cdot 10^6}}{4 \cdot \pi \cdot 10}\right)$$

Because: $P_G = P_{r_{dBm}} - P_{t_{dBm}} \Leftrightarrow PL = P_{t_{dBm}} - P_{r_{dBm}}$

$$PL = 52.1789 \quad \Rightarrow \quad K = 52.1789$$

The $PL \Rightarrow K$ because K is a constant that depends on the antenna characteristics and free space path loss up to distance d_0 [51, p. 26]:

$$M_{model}(d_i) = K - 10 \cdot \gamma \cdot log_{10}\left(\frac{d_i}{d_0}\right)$$

Where d_i is the distance for each iteration of the Table 11 and $K = 52.1789$. For the $M_{measured}$:

$$M_{measured}(d_i) = P_{t_{dBm}} - P_{r_{dBm}}$$

The measures taken on field were obtained through Received Signal Strength Indication (RSSI) and therefore the $P_{t_{dBm}}$ has to be accounted to obtain the PL. Both PL from $M_{measured}$ and M_{model} are calculated. To calculate the minimum average quadratic error value in terms to the path loss exponent γ the following equation was used:

$$F(\gamma) = \sum_{i=1}^{N_{samples}} [M_{measured}(d_i) - M_{model}(d_i)]^2$$

$$F(\gamma) = 1108.47 \cdot \gamma^2 + 6851.34 \cdot \gamma + 11035.2$$

$$\frac{\partial}{\partial \gamma} F(\gamma) = 0 \Leftrightarrow 2216.94 \cdot \gamma + 6851.34 = 0$$

$$\therefore \gamma \approx 3.1$$

The value of the path loss exponent γ for this specific environment is now defined, therefore can be applied to the Simplified Path Loss model.

$$P_r = P_t \cdot \mathrm{K} \cdot \left(\frac{d_0}{d}\right)^{3.1}$$

It's possible now to predict with accuracy the distance between nodes with the Simplified Path Loss model. In order to compare those predictions with other radio propagation models, the calculations for the Free Space propagation model are as follows:

$$P_{r_{dBm}} = P_{t_{dBm}} + G_{t_{dBi}} + G_{r_{dBi}} + 20 \cdot log_{10}\left(\frac{\lambda}{4 \cdot \pi \cdot d}\right)$$

$$P_{r_{dBm}} = 20 + 4 + 4 + 20 \cdot log_{10}\left(\frac{\frac{3 \cdot 10^8}{2437 \cdot 10^6}}{4 \cdot \pi \cdot d}\right)$$

The $P_{r_{dBm}}$ is the power received by a node according the distances in Table 11 p.92. In the Figure 51 are depicted the Field Measurements and the three radio propagation models used to predict the distance between the transmitters and the receivers.

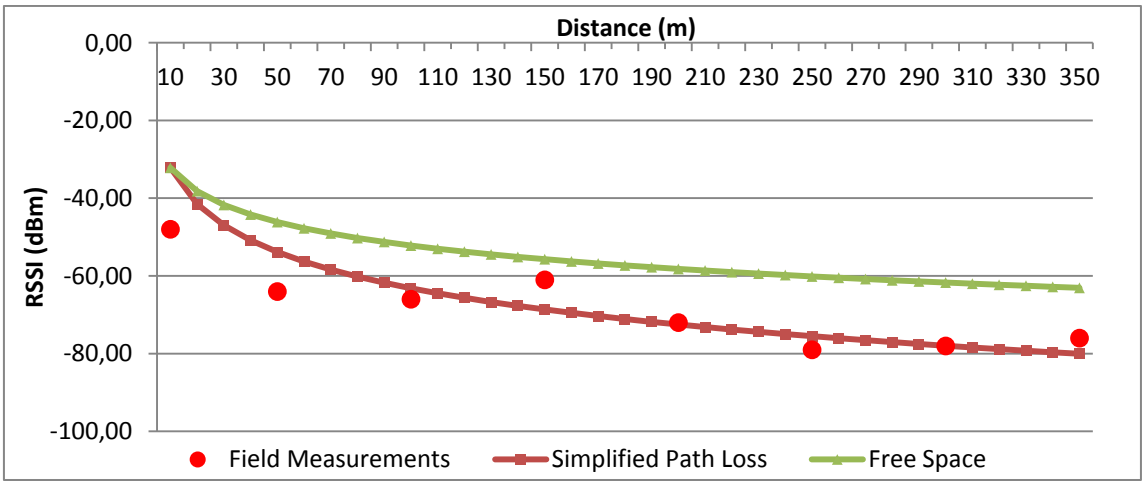

Figure 51 - Radio Propagation Models Prediction and Field Measurements

As depicted in the Figure 51 it can be concluded that with just a few measurements in the field and optimizing the path loss exponent with the Simplified Path Loss model it's possible to have an accurate prediction. The measurements obtained in the field compared with the values obtained with the Simplified Path Loss model are coherent. For instance at 200m and 300m the Simplified Path Loss prediction model has the same value as the field measurements taken.

After deploying the Multi-hop network nodes in the field, the Wi-Fi scanner tool Kismet [52], [53] described in chapter 4 p. 41 was used to capture the RSSI and Global Positioning System (GPS) of the nodes. The Kismet application associates the RSSI from the broadcasted Service Set Identifier (SSID) from nodes to the position where that signal was acquired as well as its strength. The Figure 52 depicts the result of the field coverage survey performed with the Wi-Fi Planner Tool application, using Kismet [52], [53]. The

heat map was generated by using the kisheat tool [52], [53], proving the wireless coverage from the Multi-Hop network nodes in this experimental setup on the field.

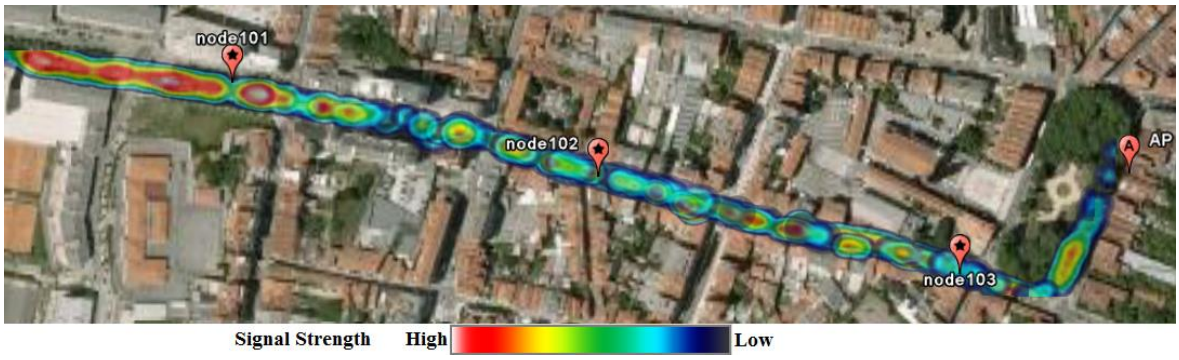

Figure 52 - Heat Map from the Experimental Multi-hop Network Coverage Setup

The heat map depicted on the Figure 52 is according to the stated distances between the nodes in the experimental Multi-hop coverage setup (section 6.1, p.90). According to the graphic from Radio Propagation Models prediction in the Figure 51, with a distance of 280m between nodes the RSSI is $-77\ dBm$. This RSSI value is comprehended within the Wi-Fi interface sensitivity according to the manufacturer [74]. The heat map is consistent with the predictions. However, due to the additional attenuations between nodes, caused by traffic (especially buses and trucks) along the street and intersections, the nodes connectivity starts to get unstable. Therefore, although the Multi-hop network has full coverage from end-to-end, it can be unstable depending on the traffic conditions. To solve this issue, a fourth node can be deployed. With a fourth node, the nodes can be apart from each other by 190m of distance, leading to a RSSI of $-72\ dBm$ according the graphic from Radio Propagation Models prediction in the Figure 51.

6.1.2. ACTIVE ROUTE

To prove the Multi-hop concept a "traceroute" command was issued from node101 to the Access Point (AP) Service Set Identifier (SSID): "Wi-Fi Porto Digital" as depicted in Figure 53.

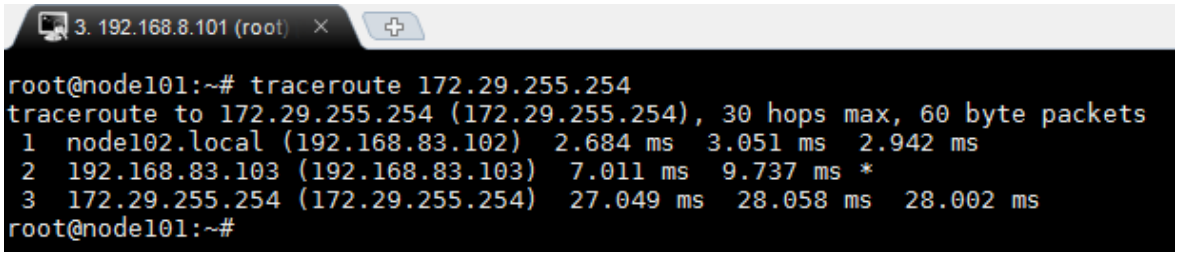

Figure 53 - Multi-hop Network traceroute to AP Wi-Fi Porto Digital

The "traceroute" test was performed while connected to node101 (192.168.83.101) via Secure Shell (SSH). The Optimized Link State Routing Protocol (OLSR) was used for this test. As depicted in the Figure 53 the next hop is the node102 (192.168.83.102) that relays the packets to the next node. From node102 the next hop is the node103 (192.168.83.103) that holds the two interfaces Independent Basic Service Set (IBSS) and station (STA). The node103 is the gateway of the Multi-hop network that in turn, forward the packets from the IBSS (wlan0) interface to the STA interface (wlan1). From the node103 (172.29.8.148) the next hop is the AP gateway (172.29.255.254) from Wi-Fi *Porto Digital*.

6.2. MULTI-HOP NETWORK PERFORMANCE SETUP

To evaluate the Multi-hop network performance at least 3 batteries were needed, one for each Multi-hop network node. Due to the lack of batteries and logistics support to place all the nodes where the coverage survey took place, the alternative was to move this setup to another place. Therefore, the Multi-hop network performance setup took place in a suburban scenario, on the outskirts of Porto city, as depicted in the Figure 54. This setup was set in order to replicate the Multi-hop coverage setup. To implement the Multi-hop network performance setup, 3 cars were used to supply power to each Multi-hop network node. According the Figure 54, the 1st node on this setup is the node located on point "B" and the 3rd node is the gateway node located on point "A". The 1st node is 230m distant from the 2nd node and the 2nd node is 200m distant from the 3rd node. The deployment of this network seeks to obtain the performance of the Optimized Link State Routing Protocol (OLSR) and *Babel* routing protocol.

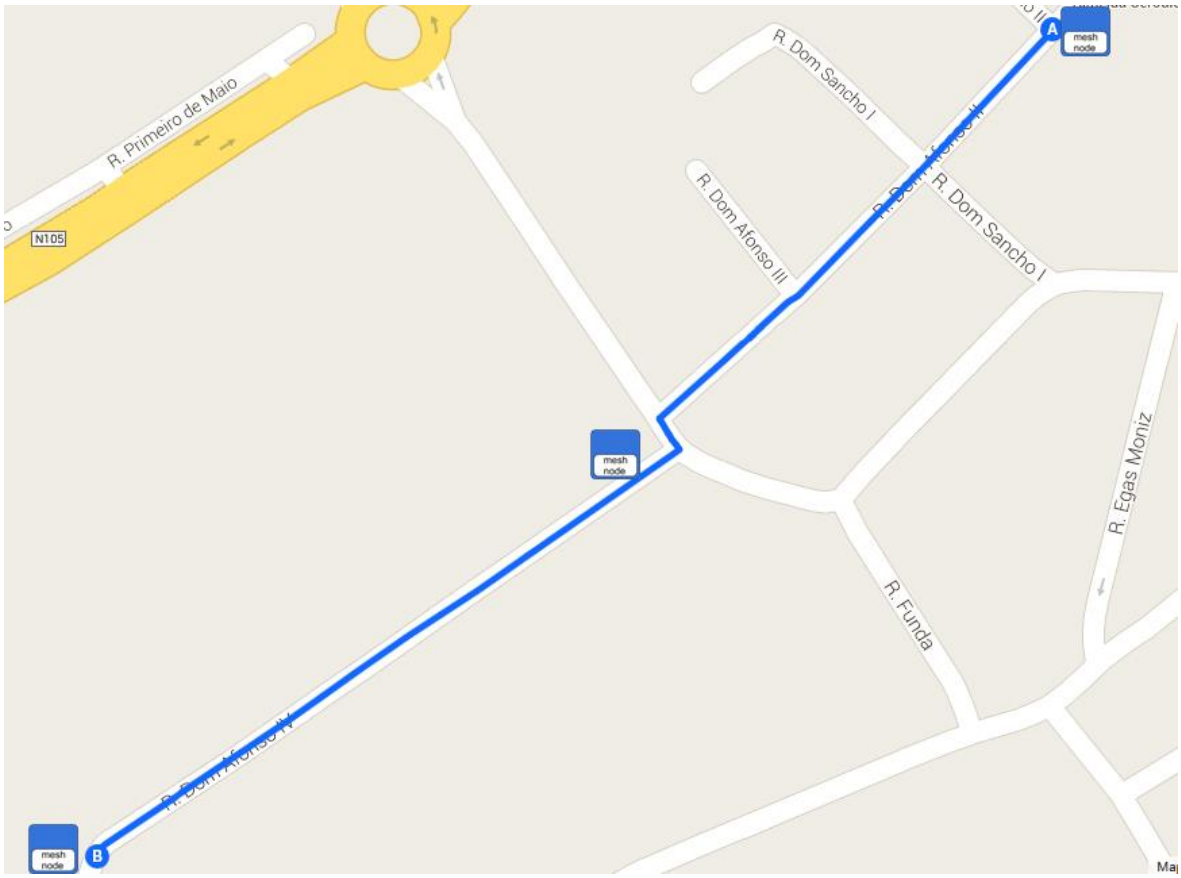

Figure 54 - Multi-hop network performance setup

To perform the tests in the Multi-hop network, an end-to-end test file with 20Mb size was created and then transferred through a Transmission Control Protocol (TCP) connection using Secure Copy Protocol (SCP). The test file was transferred using both routing protocols, first the OLSR and then the *Babel*. While the file was being tranferred, the nodes were also performing a "ping" to the gateway node. The tests were performed during two distinct situations. The "Startup" situation that is when nodes boot up and the Multi-hop network is being formed and the "Steady" situation that is when the Multi-hop network is already established. The packets exchanged in the Multi-hop network as well as the transferred file were captured using "tshark" with a time length of 300s.

6.2.1. THROUGHPUT RESULTS

The throughput results from the Multi-hop network setup were done in "Startup" and "Steady" situations, using the Optimized Link State Routing Protocol (OLSR) and *Babel* routing protocol, respectively. The graphic from Figure 55 shows that OLSR has a better throughput in "Startup" situation, however in a "Steady" situation it has a lower

throughput. The *Babel* routing protocol in the "Startup" situation has a lower throughput compared to OLSR, however in a "Steady" situation it keeps the throughput and compared to OLSR it has a better throughput.

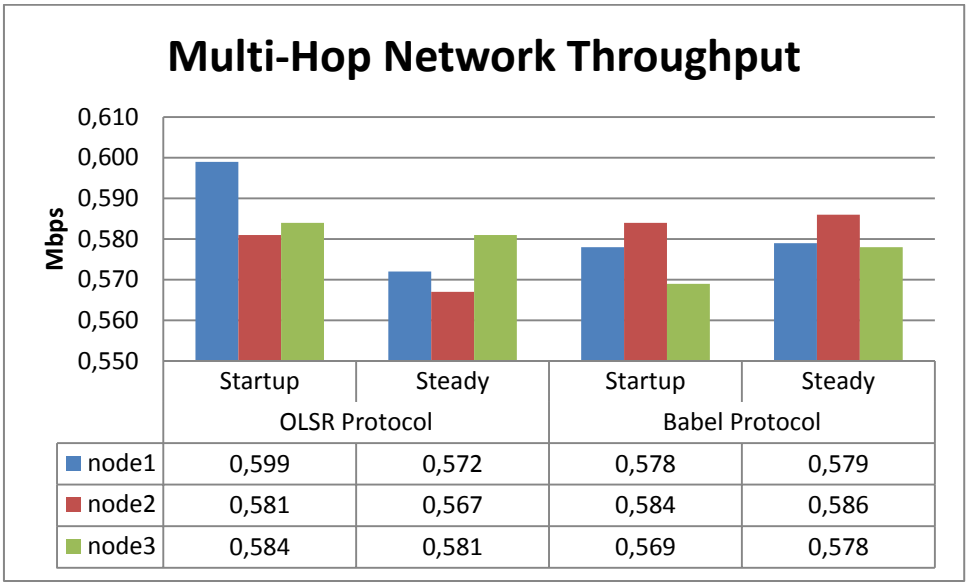

Figure 55 - Multi-hop throughput OLSR

There are two ways to improve the throughput in the Multi-hop network nodes: (1) equip the nodes with 2 physical interfaces; (2) equip the nodes with a single interface with Multiple-Input and Multiple-Output (MIMO) technology. Both ways allow the nodes to receive and transmit simultaneously [80, p. 337].

6.2.2. MULTI-HOP NETWORK PROTOCOL MESSAGES AND PACKET LOSS RESULTS

During the Multi-hop network tests all the packets exchanged were captured, including protocol messages, Optimized Link State Routing Protocol (OLSR) packets for OLSR routing protocol, *Babel* packets for *Babel* routing protocol and the Internet Control Message Protocol (ICMP) packets resulting from the "ping" command. The ICMP packets can either be replied or lost. In the Figure 56, these are referred as "ping reply" or "ping fail", respectively. This tests aim to determine the number of messages exchanged from each protocol in each "Startup" and "Steady" situations. In the tests performed in the Multi-hop network can be seen some differences between the tested protocols, especially in the amount of protocol packets exchanged in both "Startup" and "Steady" situations as well as between protocols, as depicted in the Figure 56.

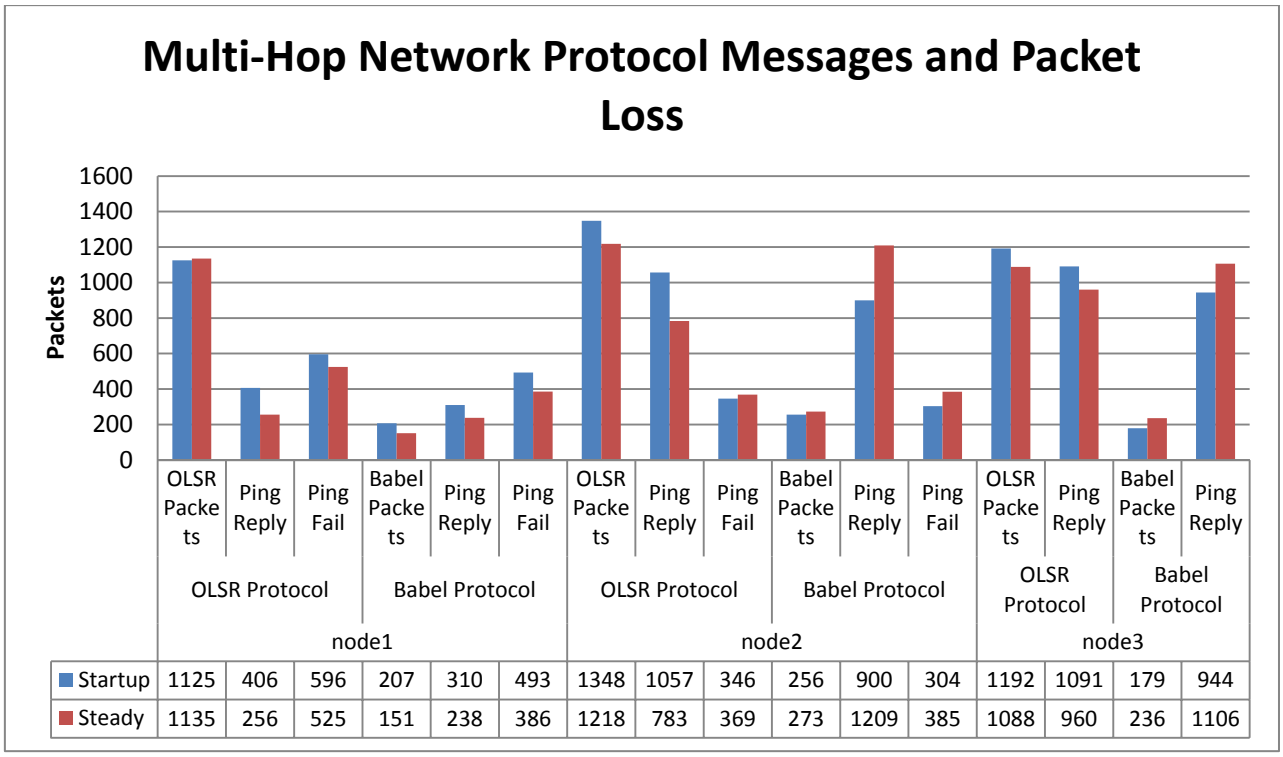

Figure 56 - Multi-hop Network Protocol Messages and Packet Loss

The results can suffer changes if OLSR and *Babel* parameters are changed. Since the Multi-hop network being tested has only 3 nodes, these results can be inconsistent due to the small amount of nodes in the network. Therefore, are also referred the results from other authors that performed field tests with the presented protocols. In the [79], "Performance Comparison of mesh routing protocols in an experimental network with bandwidth restrictions in the border router" are performed several tests up to 25 nodes in the network. As depicted in the Figure 7 of the [79], it's possible to observe that with a setup of 5 nodes the *Babel* routing protocol as a better throughput than OLSR, however this situation starts to get notorious with 15 and 25 nodes. In the [81] "Real-world performance of current proactive Multi-hop mesh protocols" the experimental setup was performed indoors with 8 nodes. These authors conclude that *Babel* offers the highest Multi-hop bandwidth and the fastest route repair time compared to OLSR. It's possible to conclude after the performance tests realized on the Multi-hop network and based on other authors tests, that *Babel* has the best performance and fastest time convergence compared with OLSR.

The Table 13 shows how the operations of installation and configuration for OLSR and *Babel* routing protocols are evaluated from easy to difficult. Being: easy - enough to read the man page and difficult - contacting the author / developer for further instructions.

Table 13 - Protocol operations, ease of use

Operations / Protocol	OLSR	*Babel*
Installation	Easy	Easy
Configuration	Easy	Difficult

The installation in both is easy since they are available in packages. The configuration for OLSR is straight forward. However in *Babel* is very difficult since the configuration file has to be created by the user. There are many issues upon boot regarding the rejected routes. Although it's stated in [70], that after 4 minutes (default), the nodes should recover routes, such thing didn't happened even with longer periods of around 12 hours, as referred in the chapter 5 p.80.

The Table 14 shows the memory usage from the tested protocols by issuing the command: `apt-get cache <service>`.

Table 14 - Memory usage by protocol

Memory / Protocol	OLSR	*Babel*
RAM	696 Kb	584 Kb
Shared Memory	528 Kb	488 Kb
Virtual Memory	2156 Kb	1920 Kb

As shown in the Table 14, the *Babel* routing protocol is the lightest of the tested protocols.

6.3. SUMMARY

This chapter describes two experiments that were conducted in the field: (1) the Experimental Coverage Setup and (2) the Multi-hop Network performance setup. In the first section is described the setup conditions and location where de Multi-hop deployment took place for the coverage survey. The coverage survey shows how the Radio Propagation Models were used to predict the nodes position. A heat map of the Multi-hop network coverage was generated and the measurements obtained in the field are compared to the calculated values obtained through the Radio Propagation models. It's proven the Multi-hop concept by showing an active route connection. In the second section is described the

Multi-hop network performance setup and are discussed the throughput results obtained from the routing protocols Optimized Link State Routing Protocol (OLSR) and *Babel*. This tests aim to determine the number of messages exchanged from each protocol in each "Startup" and "Steady" situations. They also highlight the differences between OLSR and *Babel*. Because this Multi-hop network setup has very few nodes, to support these results the work of other authors that implemented Multi-hop network on the field with 8 nodes and 25 nodes using the OLSR and *Babel*, are also referred. These authors also concluded that *Babel* offers the higher Multi-hop bandwidth and the faster route repair time than OLSR. Finally it's shown the difficulty on implementing OLSR and *Babel* as well as the space they take in the internal flash of the Raspberry Pi.

7. CONCLUSIONS

To date, the majority of Ad Hoc routing protocols research has been done using simulation only. One of the most motivating reasons to use simulation is the difficulty to recreate a real implementations. In a simulator, the code is contained within a single logical component, which is clearly defined and accessible. On the other hand, creating an implementation requires the use of a system with many components. The implementation demands an understanding not only of the routing protocols, but all the system components and their complex interactions.

This dissertation work presents a Multi-hop network designed to extend the vehicular network from Future Cities and the integration of the garbage street containers sensor into the Urban Sense platform DCUs. A *TCP Server* is also presented to retrieve GPS and Accelerometer data from vehicles.

To communicate with the *HUB* from *TNL*, this partner company provides a communication medium, RS485 bus with physical media balanced interconnecting cable. In this DCU the Raspberry Pi acts as a Master device within the *Modbus* protocol and the *HUB* act as a Slave device. To make Raspberry Pi communicate with the *HUB* through the RS485 and using *Modbus* protocol, a RS232 to RS485 converter was selected among other converters. Because the *TNL Modbus* implementation differs from the implementation available to the

Raspberry Pi platform, some adjustments had to be performed when the implementation took place. This system integrated the garbage street containers into an Urban Sense DCU. Now it's possible to keep the data of the garbage street containers of Porto city in the Urban Sense cloud DB.

The Wi-Fi Planner Tool was developed to be capable of planning a Multi-hop network, based on the location and environment conditions. This tool collects the RSSI and GPS information from the SSID: "mesh" broadcasted by the Multi-hop network nodes optimizing their location with the use of Radio Propagation Models. This tool proved to be working correctly for the purpose that was designed. The algorithm to calculate the distance is based on a "flat earth" model [57] therefore for long distances the error starts to become notorious. A specific delay between samples could be used to improve the measurements quality since reflection, diffraction, scattering and shadowing effects are not being considered.

It's possible to conclude after the performance tests realized on the Multi-hop network and also taking into consideration other authors tests, that *Babel* has the best performance and fastest time convergence compared with OLSR. However due to *Babel* rejected routes problem during the node identification (on boot), this led to node isolation in the network, creating network connection failure. The differences between *Babel* and OLSR in terms of performance are not that notorious and because OLSR is more reliable and easy to deploy compared to *Babel*, despite *Babel* better results in the overall, they are not notorious enough to discard OLSR. For the Multi-Hop setup presented and to prevent node isolation the most suitable routing protocol is OLSR.

The TCP Server developed for the OBUs proved to be working on MIPS architecture. It can be accessed by any platform exchanging the messages accordingly whitout crash reports. Upon development of the TCP Server Application the type of server or protocol for messages exchanged was not specified, therefore this application was designed to be costumized and expanded in terms of number of clients connected simultaneously, type and number of messages exchanged, as well as the way it saves and prints the data collected.

7.1. CONTRIBUTIONS AND RESULTS

This dissertation work produced contributions and results for the Future Cities Project.

The contributions within the scope of this dissertation work are:

- The implementation of the TCP Server for OBUs to collect GPS and Accelerometer data from vehicles;
- The integration of the garbage street containers sensor into a DCU for the Urban Sense Platform;
- The Multi-hop network architecture extension and its implementation;
- The development of the Wi-Fi Planner Tool to collect field measurements for the Multi-hop network planning and heatmap generator for Google Earth [2] application;
- The study for the best technology/protocol for the Multi-hop network extension to the Future Cities Project.

The results obtained with this dissertation work are:

- The Multi-Hop network as an extension to the Vehicular network;
- The Multi-Hop network as a redundant path to the public APs, RSUs and the Vehicular network;
- The best protocol choice to the presented setup;
- The Wi-Fi Planner Tool for further network planning and temperature-map style overlay for Google Earth [2] application;
- The tools and scripts developed for Multi-hop network node field deployment.

7.2. FUTURE WORK

For future work, the implementation of other technologies, such as Arduino based implementations resorting to X-Bee or LoRA to transmit the data across the Multi-hop network. With the authorization from *TNL* (partner company) the direct access to the 433MHz radio frequency, removing the *HUB* as a "middle man" and the implementation of the DCU for garbage street containers with a direct access to the sensor inside the garbage containers. Optimization of the implemented routing protocols parameters to achieve better performance on the Multi-Hop network for the urban scenario. The implementation of reactive MANET routing protocols and mesh routing protocols to compare with the *Babel* and OLSR routing protocols. Due to the difficulty faced upon *Babel* routing protocol configuration and because the solution for rejected routes problem had time implications, this lead to an unfinished IEEE 802.11s implementation. As future work in this matter could be the implementation of the IEEE 802.11s and further

implementation of WiFIX for comparison between Layer 2 protocols and the hereby presented Layer 3 protocols.

References

[1] “Future Cities » An Ecosystem for the Future.” [Online]. Available: http://futurecities.up.pt/site/. [Accessed: 11-Mar-2015].

[2] “Google Earth.” [Online]. Available: https://www.google.com/earth/. [Accessed: 09-Feb-2015].

[3] “WHO | Urban population growth,” *WHO*. [Online]. Available: http://www.who.int/gho/urban_health/situation_trends/urban_population_growth_text/en/. [Accessed: 30-Oct-2014].

[4] “Home,” *The ZigBee Alliance*. [Online]. Available: http://www.zigbee.org/. [Accessed: 09-Mar-2015].

[5] “Home.” [Online]. Available: http://lora-alliance.org/. [Accessed: 09-Mar-2015].

[6] R. S. Schwartz, R. R. R. Barbosa, N. Meratnia, G. Heijenk, and H. Scholten, “A directional data dissemination protocol for vehicular environments,” *Comput. Commun.*, vol. 34, no. 17, pp. 2057–2071, Nov. 2011.

[7] M. Torrent-Moreno, F. Schmidt-Eisenlohr, H. Fubler, and H. Hartenstein, “Effects of a realistic channel model on packet forwarding in vehicular ad hoc networks,” in *Wireless Communications and Networking Conference, 2006. WCNC 2006. IEEE*, 2006, vol. 1, pp. 385–391.

[8] J. Ott, D. Kutscher, and C. Dwertmann, “Integrating DTN and MANET routing,” in *Proceedings of the 2006 SIGCOMM workshop on Challenged networks*, 2006, pp. 221–228.

[9] K. L. Scott and S. Burleigh, “Bundle Protocol Specification.” [Online]. Available: http://tools.ietf.org/html/rfc5050. [Accessed: 25-Nov-2014].

[10] A. Lindgren, E. Davies, S. Grasic, and A. Doria, “Probabilistic Routing Protocol for Intermittently Connected Networks.” [Online]. Available: http://tools.ietf.org/html/rfc6693. [Accessed: 20-Mar-2015].

[11] “IBR-DTN API.” [Online]. Available: https://trac.ibr.cs.tu-bs.de/project-cm-2012-ibrdtn/raw-attachment/wiki/docs/api/api.pdf. [Accessed: 25-Nov-2014].

[12] M. Doering, S. Lahde, J. Morgenroth, and L. Wolf, “IBR-DTN: an efficient implementation for embedded systems,” in *Proceedings of the third ACM workshop on Challenged networks*, 2008, pp. 117–120.

[13] “IBR-DTN.” [Online]. Available: http://trac.ibr.cs.tu-bs.de/project-cm-2012-ibrdtn. [Accessed: 25-Nov-2014].

[14] P. Basanta-Val, M. García-Valls, and M. Baza-Cuñado, “A Simple Data Mulling Protocol (DRAFT).”

[15] G. Anastasi, M. Conti, and M. Di Francesco, “Data collection in sensor networks with data mules: An integrated simulation analysis,” in *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on*, 2008, pp. 1096–1102.

[16] R. C. Shah, S. Roy, S. Jain, and W. Brunette, “Data MULEs: modeling a three-tier architecture for sparse sensor networks,” *Sens. Netw. Protoc. Appl. 2003 Proc. First IEEE 2003 IEEE Int. Workshop On*, pp. 30–41, May 2003.

[17] L. Pelusi, A. Passarella, and M. Conti, “Opportunistic networking: data forwarding in disconnected mobile ad hoc networks,” *Commun. Mag. IEEE*, vol. 44, no. 11, pp. 134–141, 2006.

[18] G. Anastasi, M. Conti, A. Passarella, and L. Pelusi, “Mobile-relay forwarding in opportunistic networks,” *Adapt. Cross Layer Des. Wirel. Netw.*, 2008.

[19] M. R. Schurgot, C. Comaniciu, and K. Jaffres-Runser, “Beyond traditional DTN routing: social networks for opportunistic communication,” *ArXiv Prepr. ArXiv11102480*, 2011.

[20] V. F. S. Mota, F. D. Cunha, D. F. Macedo, J. M. S. Nogueira, and A. A. F. Loureiro, “Protocols, mobility models and tools in opportunistic networks: A survey,” *Comput. Commun.*, vol. 48, pp. 5–19, Jul. 2014.

[21] D. J. Dubois, Y. Bando, K. Watanabe, and H. Holtzman, “Lightweight self-organizing reconfiguration of opportunistic infrastructure-mode WiFi networks,” in *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, 2013, pp. 247–256.

[22] E. Rozner, J. Seshadri, Y. Mehta, and L. Qiu, “Simple opportunistic routing protocol for wireless mesh networks,” in *Wireless Mesh Networks, 2006. WiMesh 2006. 2nd IEEE Workshop on*, 2006, pp. 48–54.

[23] E. Rozner, J. Seshadri, Y. Mehta, and L. Qiu, “SOAR: Simple opportunistic adaptive routing protocol for wireless mesh networks,” *Mob. Comput. IEEE Trans. On*, vol. 8, no. 12, pp. 1622–1635, 2009.

[24] Y. Yuan, H. Yang, S. H. Wong, S. Lu, and W. Arbaugh, “ROMER: resilient opportunistic mesh routing for wireless mesh networks,” in *IEEE workshop on wireless mesh networks (WiMesh)*, 2005, vol. 12.

[25] “Citoyens Capteurs - Des technologies extraordinaires pour des gens ordinaires.” .

[26] “CITI-SENSE > Home.” [Online]. Available: http://www.citi-sense.eu/. [Accessed: 17-Feb-2015].

[27] “Libelium - Smart City project in Serbia to monitor Environmental Parameters by Public Transportation with Waspmote | Libelium.” [Online]. Available: http://www.libelium.com/smart_city_environmental_parameters_public_transportation_waspmote/. [Accessed: 17-Feb-2015].

[28] “EDP Ingeniería.” [Online]. Available: http://www.edpingenieria.com/portfolio.html#. [Accessed: 17-Feb-2015].

[29] M. Attack, “Realtek Wi-Fi Direct Programming Guide - Dishing Tech.” .

[30] “Thinktube - WiFi Direct.” [Online]. Available: http://www.thinktube.com/tech/android/wifi-direct. [Accessed: 16-Jan-2015].

[31] R. Campos, R. Duarte, F. Sousa, M. Ricardo, and J. Ruela, “Network infrastructure extension using 802.1D-based wireless mesh networks,” *Wirel. Commun. Mob. Comput.*, vol. 11, no. 1, pp. 67–89, Jan. 2011.

[32] “Categorization and Selection of Routing Protocols for Wireless Mesh Network.” [Online]. Available: http://www.academia.edu/10119559/Categorization_and_Selection_of_Routing_Protocols_for_Wireless_Mesh_Network. [Accessed: 20-Feb-2015].

[33] S. R. Das, E. M. Belding-Royer, and C. E. Perkins, “Ad hoc On-Demand Distance Vector (AODV) Routing.” [Online]. Available: http://tools.ietf.org/html/rfc3561. [Accessed: 04-Nov-2014].

[34] “Routing Protocols in MANETs,” *Eexploria*. .

[35] P. J. <philippe.jacquet@inria.fr>, “Optimized Link State Routing Protocol (OLSR).” [Online]. Available: http://tools.ietf.org/html/rfc3626. [Accessed: 06-Nov-2014].

[36] “www.olsr.org | an adhoc wireless mesh routing daemon.” [Online]. Available: http://www.olsr.org/. [Accessed: 06-Nov-2014].

[37] “Proactive protocols - OLSR.” [Online]. Available: http://www.olsr.org/docs/report_html/node17.html. [Accessed: 06-Nov-2014].

[38] U. Herberg, T. Clausen, P. Jacquet, and C. Dearlove, “The Optimized Link State Routing Protocol Version 2.” [Online]. Available: http://tools.ietf.org/html/rfc7181. [Accessed: 05-Nov-2014].

[39] J. C. <jch@pps.jussieu.fr>, “The Babel Routing Protocol.” [Online]. Available: http://tools.ietf.org/html/rfc6126. [Accessed: 04-Nov-2014].

[40] C. E. Perkins and P. Bhagwat, “Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers,” in *ACM SIGCOMM Computer Communication Review*, 1994, vol. 24, pp. 234–244.

[41] C. Aichele, S. Wunderlich, A. Neumann, and M. Lindner, “Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.).” [Online]. Available: https://tools.ietf.org/html/draft-wunderlich-openmesh-manet-routing-00#section-3. [Accessed: 13-Jan-2015].

[42] “Better Approach To Mobile Ad hoc Networking (B.A.T.M.A.N.) | DES-Testbed.” [Online]. Available: http://www.des-testbed.net/content/better-approach-mobile-ad-hoc-networking-batman. [Accessed: 17-Jan-2015].

[43] “11-06-1778-01-000s-hwmp-specification.doc.” [Online]. Available: https://mentor.ieee.org/802.11/dcn/06/11-06-1778-01-000s-hwmp-specification.doc. [Accessed: 12-Nov-2014].

[44] “Wireless Mesh Networks in the IEEE LMSC: ComNets Research Group.” [Online]. Available: http://www.comnets.rwth-aachen.de/Hi-GMC-200.5835.0.html. [Accessed: 19-Nov-2014].

[45] D. Harkins, “Simultaneous Authentication of Equals: A Secure, Password-Based Key Exchange for Mesh Networks,” 2008, pp. 839–844.

[46] Q.-A. Minhas, H. Mahmood, and H. Malik, “The role of ad hoc networks in mobile telecommunication,” *RECENT Dev. Mob. Commun. Multidiscip. APPROACH*, vol. 179, 2011.

[47] E. A. Panaousis, G. Drew, G. P. Millar, T. A. Ramrekha, and C. Politis, “A Test-Bed Implementation for Securing OLSR In Mobile Ad-Hoc Networks,” *Int. J. Netw. Secur. Its Appl.*, vol. 2, no. 4, pp. 143–162, Oct. 2010.

[48] P. Rane, “Design of Modbus Controller Using VHDL for Remote Administrations of a Network of Devices,” in *Emerging Trends in Engineering and Technology (ICETET), 2010 3rd International Conference on*, 2010, pp. 694–697.

[49] “Simply Modbus - Data Communication Test Software - Modbus ASCII vs RTU.” [Online]. Available: http://www.simplymodbus.ca/TCP.htm. [Accessed: 03-Feb-2015].

[50] A. Mitra, “Lecture Notes on Mobile Communication,” *Curric. Dev. Cell Proj. QIP IIT Guwahati*, 2009.

[51] A. Goldsmith, “Search eLibrary.”

[52] “Kismet.” [Online]. Available: https://www.kismetwireless.net/index.shtml. [Accessed: 26-Jan-2015].

[53] “kisheat - Google Earth temperature maps from kismet data - Google Project Hosting.” [Online]. Available: https://code.google.com/p/kisheat/. [Accessed: 26-Jan-2015].

[54] “Microsoft Word - TM1638English version.doc - tm1638.pdf.” [Online]. Available: http://cholla.mmto.org/computers/avr/cool_parts/tm1638.pdf. [Accessed: 10-Mar-2015].

[55] “Martin’s Atelier: The TM1638 & the Raspberry Pi.” [Online]. Available: http://www.mjoldfield.com/atelier/2012/08/pi-tm1638.html. [Accessed: 04-Feb-2015].

[56] “Compatible GPSes.” [Online]. Available: http://catb.org/gpsd/hardware.html. [Accessed: 06-Feb-2015].

[57] “Calculate distance and bearing between two Latitude/Longitude points using haversine formula in JavaScript.” [Online]. Available: http://www.movable-type.co.uk/scripts/latlong.html. [Accessed: 10-Feb-2015].

[58] D. E. ERICKSEN, “DISTANCE AND BEARING CALCULATIONS.”

[59] “How to Choose the Right Platform: Raspberry Pi or BeagleBone Black?,” *MAKE*. .

[60] “Intel Galileo vs. Raspberry Pi | Mouser.” [Online]. Available: http://pt.mouser.com/applications/open-source-hardware-galileo-pi/. [Accessed: 29-Jan-2015].

[61] “ODROID | Hardkernel.” [Online]. Available: http://www.hardkernel.com/main/products/prdt_info.php. [Accessed: 30-Jan-2015].

[62] “Arduino vs. Raspberry Pi vs. CubieBoard vs. Gooseberry vs. APC Rock vs. OLinuXino vs. Hackberry A10 | Keeward’s TechWatch blog.” [Online]. Available: http://techwatch.keeward.com/geeks-and-nerds/arduino-vs-raspberry-pi-vs-cubieboard-vs-gooseberry-vs-apc-rock-vs-olinuxino-vs-hackberry-a10/. [Accessed: 30-Jan-2015].

[63] “0900766b81249c2c.pdf.” [Online]. Available: http://docs-europe.electrocomponents.com/webdocs/1249/0900766b81249c2c.pdf. [Accessed: 31-Jan-2015].

[64] “DS_USB_RS485_PCB.pdf.” [Online]. Available: http://www.ftdichip.com/Support/Documents/DataSheets/Cables/DS_USB_RS485_PCB.pdf. [Accessed: 31-Jan-2015].

[65] “SCH_IM130710001.pdf.” [Online]. Available: http://inmotion.pt/documentation/others/INM-0815/SCH_IM130710001.pdf. [Accessed: 31-Jan-2015].

[66] “WiringPi.” .

[67] “libmodbus.org - Download.” [Online]. Available: http://libmodbus.org/download/. [Accessed: 27-Feb-2015].

[68] “babeld 0.96 - files.html.” [Online]. Available: http://sourcecodebrowser.com/babeld/0.96/files.html. [Accessed: 19-Nov-2014].

[69] “brunosfer/MSc-FutureCities,” *GitHub*. [Online]. Available: https://github.com/brunosfer/MSc-FutureCities. [Accessed: 24-Mar-2015].

[70] “BABELD(8) manual page.” [Online]. Available: http://www.pps.univ-paris-diderot.fr/~jch/software/babel/babeld.html. [Accessed: 06-Mar-2015].

[71] “sivel/speedtest-cli,” *GitHub*. [Online]. Available: https://github.com/sivel/speedtest-cli. [Accessed: 24-Feb-2015].

[72] “OpenWrt.” [Online]. Available: https://openwrt.org/. [Accessed: 14-Mar-2015].

[73] “OpenWrt Downloads.” [Online]. Available: http://downloads.openwrt.org/. [Accessed: 30-Apr-2015].

[74] "- TL-WN821N(C_V3_User_Guide.pdf." [Online]. Available: http://www.tp-link.pt/Resources/document/TL-WN821N(C_V3_User_Guide.pdf. [Accessed: 22-Jan-2015].

[75] S. S. Jain and others, "Analysis of OLSR, DSR, DYMO routing protocols in mobile Ad-Hoc Networks using OMNeT++ Simulation," *Glob. J. Comput. Sci. Technol.*, vol. 14, no. 1, 2014.

[76] R. Malekian, A. Karadimce, and A. H. Abdullah, "AODV and OLSR Routing Protocols in MANET," 2013, pp. 286–289.

[77] E. Medina, R. Meseguer, C. Molina, and D. Royo, "OLSRp: Predicting control information to achieve scalability in OLSR ad hoc networks," in *Mobile Networks and Management*, Springer, 2011, pp. 225–236.

[78] W. Yan, W. Guo, and J. Liu, "An implementation and study of OLSR protocol in linux OS," in *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM'08. 4th International Conference on*, 2008, pp. 1–4.

[79] M. E. VILLAPOL, D. PÉREZ ABREU, C. BALDERAMA, and M. COLOMBO, "PERFORMANCE COMPARISON OF MESH ROUTING PROTOCOLS IN AN EXPERIMENTAL NETWORK WITH BANDWIDTH RESTRICTIONS IN THE BORDER ROUTER," *Rev. Fac. Ing. Univ. Cent. Venezuela*, vol. 28, no. 1, pp. 7–14, 2013.

[80] Y. Xiao, *Security in distributed, grid, mobile, and pervasive computing*. Boca Raton: Auerbach Publications, 2007.

[81] M. Abolhasan, B. Hagelstein, and J. C.-P. Wang, "Real-world performance of current proactive multi-hop mesh protocols," *Commun. 2009 APCC 2009 15th Asia-Pac. Conf. On*, pp. 44–47, Oct. 2009.

Appendix A

In this appendix is presented the Makefile developed to Cross-Compile the OBU TCP Server Application.

```
# Author: Bruno Fernandes
# Makefile configurations (OpenWrt)
CFLAGS = -O2 -Wall --pedantic -Wno-write-strings -Wno-long-long
LIBS = -lrt -lpthread
SOURCES = tcpServerMIPS.c gps.c accelerometer.c
OBJECTS=$(SOURCES:.c=.o)
EXECUTABLE = tcpservermips
HOSTROOT = /home/bruno/OpenWrt/trunk/staging_dir/toolchain-mips_34kc_gcc-
4.8-linaro_uClibc-0.9.33.2/
CC = $(HOSTROOT)/bin/mips-openwrt-linux-gcc
CPP = $(HOSTROOT)/bin/mips-openwrt-linux-g++
STAGING_DIR = /home/bruno/OpenWrt/trunk/staging_dir/
export STAGING_DIR

all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LIBS) $(OBJECTS) -o $@

%.o: %.c
	$(CC) -c $(CFLAGS) $< -o $@

clean:

	rm -rf *o $(EXECUTABLE)
```

Appendix B

The code presented in this appendix is the Application developed for the DCU for garbage street containers.

```
//Author: Bruno Fernandes
//Data Collector Unit For Garbage Street Containers
// Revised: 28-01-2015

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <time.h>
#include <stdio.h>
#include <modbus.h>
#include <modbus-rtu.h>
#include <errno.h>
#include <unistd.h>
#include <sys/timeb.h>
#include <bcm2835.h>

#define BCM2835_PERI_BASE 0x20000000
#define BCM2835_GPIO_BASE (BCM2835_PERI_BASE + 0x200000)
#define PAGE_SIZE (4*1024)
#define BLOCK_SIZE (4*1024)
#define DCU_LOGFILE_PATH "/home/pi/Programs/dcu_gsc/dcu_gsc.log"

int mem_fd;
void *gpio_map;
volatile unsigned *gpio;
uint8_t raw_req[] = {0x01, 0x03, 0x00, 0x00, 0x00, 0x05};
uint8_t rsp[MODBUS_RTU_MAX_ADU_LENGTH];
int baud_rate, par, data_bits, stop_bits, adr;
/*Set up arrays with the different possibilities for communications
parameters */
int baud_rates[12] = {110, 300, 600, 1200, 2400, 4800, 9600, 19200,
38400, 57600, 115200, 230400};
char parity[3] = {'N', 'E', 'O'};
int databits[2] ={7, 8};
int stopbits[2] = {1, 2};
/*set up a struct for the positive responses from the discovery
sequence*/
struct discover{
int baud; //Baud rate
char par; //Parity
int db; //No. of data bits
int stopbits; //No. of stop bits
int adr; //The modbus address
};
struct discover st_discover[20]; //allow 20 positive responses for now
int index_discover;

int discover(void);
```

```
void setup_io(void);
int getData(void);

int main(int argc, char **argv)
{

//if (bcm2835_init()) printf("bcm2835 Iniciated, returned: %d\n\n",
bcm2835_init());  //For DEBUG
bcm2835_init();
setup_io();

bcm2835_gpio_fsel(RPI_GPIO_P1_11, BCM2835_GPIO_FSEL_ALT3);   //First
argument is PIN17. (Yes P1_11 it's PIN17) the second argument it's to
activate the RTS/CTS function for RS485.
int j=0;
discover();
do{ j = getData();
}while(j != 0);
return 0;
}

void setup_io()
{
/* open /dev/mem */
if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
printf("can't open /dev/mem \n");
exit(-1);
}
/* mmap GPIO */
gpio_map = mmap(
NULL, //Any adddress in our space will do
BLOCK_SIZE, //Map length
PROT_READ|PROT_WRITE,// Enable reading & writing to mapped memory
MAP_SHARED, //Shared with other processes
mem_fd, //File to map
BCM2835_GPIO_BASE //Offset to GPIO peripheral
);
close(mem_fd); //No need to keep mem_fd open after mmap
if (gpio_map == MAP_FAILED) {
printf("mmap error %d\n", (int)gpio_map);//errno also set!
exit(-1);
}
gpio = (volatile unsigned *)gpio_map;
}

int discover()
{
modbus_t *mb;
uint16_t tab_reg[32];
int rr; /*read registers*/
int i;
index_discover =0;

for (baud_rate = 4; baud_rate <= 10; baud_rate++) //We'll only use
baudrates from 2400 to 115200
{
for (par = 0; par <= 2; par++)
{
for (data_bits = 0; data_bits <= 1; data_bits++)
{
for (stop_bits = 0; stop_bits <= 1; stop_bits++)
```

```
{
for (adr = 64; adr <= 64; adr++)
{
mb = modbus_new_rtu("/dev/ttyAMA0", baud_rates[baud_rate], parity[par],
databits[data_bits], stopbits[stop_bits]);

modbus_set_slave(mb,adr);
modbus_rtu_set_serial_mode(mb, MODBUS_RTU_RS485);     //RS485
modbus_rtu_set_rts(mb, MODBUS_RTU_RTS_UP);
modbus_connect(mb);
rr = modbus_read_registers(mb, 7, 1, tab_reg);  //mb, holdingRegister
address, numberOfAdresses
modbus_rtu_set_rts(mb, MODBUS_RTU_RTS_DOWN);

if (rr != -1)
{
st_discover[index_discover].baud = baud_rates[baud_rate];
st_discover[index_discover].par = parity[par];
st_discover[index_discover].db = databits[data_bits];
st_discover[index_discover].stopbits = stopbits[stop_bits];
st_discover[index_discover].adr = adr;
index_discover++;
}
modbus_close(mb);
modbus_free(mb);
}
}
}
}
}

for (i=0; i< index_discover; i++)
{
printf("found device at address %i, Baudrate %i, data bits %i, parity %c,
stop bits %i\n", st_discover[i].adr, st_discover[i].baud,
st_discover[i].db, st_discover[i].par, st_discover[i].stopbits);
}
return 0;
}

int getData()
{
FILE *fp;
time_t ltime; /* calendar time */
ltime=time(NULL); /* get current cal time */
modbus_t *mb;
uint16_t tab_reg[32];
int rr; /*read registers*/
//int wr; /*write registers */

//int i;
int f;
int32_t lat;
int32_t lon;
float flat;
float flon;
char clat[10];
char clon[10];

mb = modbus_new_rtu("/dev/ttyAMA0", 19200, 'N', 8, 1); //Settings
required by TNL
```

```
if (mb == NULL)
{
fprintf(stderr, "Unable to create the libmodbus context\n");
return -1;
}
modbus_set_debug(mb,FALSE); //Enable debugging for tests
if (modbus_set_slave(mb,64)== -1)
{
fprintf(stderr, "set slave failed: %s\n", modbus_strerror(errno));
modbus_free(mb);
return -2;
}
modbus_rtu_set_serial_mode(mb, MODBUS_RTU_RS485);     //RS485
modbus_rtu_set_rts(mb, MODBUS_RTU_RTS_UP);
if (modbus_connect(mb) == -1)
{
fprintf(stderr, "Connection failed: %s\n", modbus_strerror(errno));
modbus_free(mb);
return -3;
}
// In this cicle I can set the number of samples I can collect in each
request.
for (f=0; f<1; f++)
{
modbus_rtu_set_rts(mb, MODBUS_RTU_RTS_UP);
rr = modbus_read_registers(mb, 0, 17, tab_reg); //Read from 0 to 16
registers
modbus_rtu_set_rts(mb, MODBUS_RTU_RTS_DOWN);
if (rr == -1)
{
fprintf(stderr, "Couldn't connect %s\n", modbus_strerror(errno));
return -4;
}
//FOR DEBUG ONLY Prints RAW Registers and Values and Prints Registers in
Holding Reg format
/*

for (i=0; i < rr; i++)
{
printf(" reg[%d]=%d (0x%X)\n", i, tab_reg[i], tab_reg[i]);
}
*/
}
/*
for (f=0; f<rr; f++) //print out the registers
{
printf("\nRegister 400%i value: %i", f, tab_reg[f]);
}
*/
// Merge POS_LAT_H with POS_LAT_L and POS_LON_H with POS_LON_L
lat = tab_reg[11] << 12;
lat += tab_reg[12];
lon = tab_reg[13] << 16;
lon += tab_reg[14];
lon = ~lon;
sprintf(clat,"%u", lat);
sprintf(clon,"%u", lon);
flat = atof(clat);
flat = flat/1000000;
flon = atof(clon);
```

```
flon = flon/-1000000;
//Save results on a log file
fp = fopen(DCU_LOGFILE_PATH, "a+");
fprintf(fp, "\nTime: %s\n", asctime(localtime(&ltime)));
fprintf(fp, "UUI0: %i\n", tab_reg[0]);
fprintf(fp, "UUI1: %i\n", tab_reg[1]);
fprintf(fp, "UUI2: %i\n", tab_reg[2]);
fprintf(fp, "UUI3: %i\n", tab_reg[3]);
fprintf(fp, "UUI4: %i\n", tab_reg[4]);
fprintf(fp, "UUI5: %i\n", tab_reg[5]);
fprintf(fp, "DIST: %i\n", tab_reg[6]);
fprintf(fp, "ACCX: %i\n", tab_reg[7]);
fprintf(fp, "ACCY: %i\n", tab_reg[8]);
fprintf(fp, "ACCZ: %i\n", tab_reg[9]);
fprintf(fp, "TEMP: %i\n", tab_reg[10]);
fprintf(fp, "PLAT: %.6f\n", flat);
fprintf(fp, "PLON: %.6f\n", flon);
fprintf(fp, "ACT: %i\n", tab_reg[15]);
fprintf(fp, "BAT: %i\n\n", tab_reg[16]);
fclose(fp);
// Print out results on the console
printf("\n\nTime:\t %s\n", asctime(localtime(&ltime)));
printf("UUI0:\t %i\n", tab_reg[0]);
printf("UUI1:\t %i\n", tab_reg[1]);
printf("UUI2:\t %i\n", tab_reg[2]);
printf("UUI3:\t %i\n", tab_reg[3]);
printf("UUI4:\t %i\n", tab_reg[4]);
printf("UUI5:\t %i\n", tab_reg[5]);
printf("DIST:\t %i\n", tab_reg[6]);
printf("ACCX:\t %i\n", tab_reg[7]);
printf("ACCY:\t %i\n", tab_reg[8]);
printf("ACCZ:\t %i\n", tab_reg[9]);
printf("TEMP:\t %i\n", tab_reg[10]);
printf("PLAT:\t %.6f\n", flat);
printf("PLON:\t %.6f\n", flon);
printf("ACT:\t %i\n", tab_reg[15]);
printf("BAT:\t %i\n\n", tab_reg[16]);
modbus_close(mb);
modbus_free(mb);
return 0;
}
```

Appendix C

This appendix contains the source code of the *pingLED* script. It's the same for every network nodes, except the gateway. The gateway pings an address, i.e. 'www.google.com'.

```
#!/bin/bash
#pingLED.sh
IP='192.168.83.103'
ATTEMPTS='10000'
#Functions to control Raspberry Pi ACT internal LED
on() {
      sudo echo 1 > /sys/class/leds/led0/brightness
}
off() {
      sudo echo 0 > /sys/class/leds/led0/brightness
}
deactivate() {
      sudo echo none >/sys/class/leds/led0/trigger
}
default() {
      sudo echo mmc0 >/sys/class/leds/led0/trigger
}
quit(){
      default
      exit 1
}
# Capture CTRL+C, CTRL+Z, Broken Pipe and quit
trap 'quit' SIGINT SIGQUIT SIGTERM SIGHUP SIGKILL SIGPIPE
trap '' SIGTSTP
deactivate
for (( i=0; i<=$ATTEMPTS; i++ ))
do
sudo ping -c4 $IP > /dev/null 2>&1

if [ $? -eq 0 ]; then
      on
      i=0
else
      off
fi
IPERF_PROC=`pgrep iperf`
if [ "${IPERF_PROC:-null}" != null ]; then
      quit
fi
#echo "Number of trials: $i"
done
# Reset all traps
trap - SIGTSTP SIGINT SIGQUIT SIGTERM SIGHUP SIGKILL SIGPIPE
default
echo " "
echo "All Done!"
exit 0
```

Appendix D

The code presented in this appendix is the Wi-Fi Planner Tool Application.

```
/**
 * wifiPlannerTool
 * Author: Bruno Fernandes
 * Revised: 02/02/2015
**/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <bcm2835.h>
#include "tm1638.h"

#define WIFI_SCAN_LOGFILE_PATH
"/home/pi/Programs/wifiPlannerTool/wifiPlannerTool.log"
#define CSV_FILE_PATH
"/home/pi/Programs/wifiPlannerTool/wifiPlannerTool.csv"
#define START_KISMET "/home/pi/Programs/gps/gpsKismet.start"
#define START_KISMET_CLIENT "sudo kismet_client -s"
#define STOP_KISMET "/home/pi/Programs/gps/gpsKismet.stop"
#define GPS "/home/pi/Programs/gps/gps.start"
#define SAMPLES 5
#define S1 128
#define S2 64
#define S3 32
#define S4 16
#define S5 8
#define S6 4
#define S7 2
#define S8 1


char gps[10];
char kismets[10];
char kismetc[10];
char gpsdata[80];
int lastchar;
uint8_t x=0;
float average=0;
float variance=0;
float std_dev=0;
bool wi_ret = 0;
bool flag=0;
bool flag_csv=0;
float lt1, ln1, lt2, ln2;
float distance;
float gpslatlon[4];
```

```
int gps_detector(void);
int wifi_scanner(int samples);
int kismet_server_detector(void);
int kismet_client_detector(void);
int kismet_start(void);
int kismet_stop(void);
int rpi_reboot(void);
int rpi_shutdown(void);
float calcDistance(float lat1, float lon1, float lat2, float lon2);
float gps_data(void);
void make_csv(void);


int main(int argc, char *argv[])
{
  tm1638_p t;

  if (!bcm2835_init())
    {
      printf("Unable to initialize BCM library\n");
      return -1;
    }

  t = tm1638_alloc(17, 27, 22);
  if (!t)
    {
      printf("Unable to allocate TM1638\n");
      return -2;
    }

tm1638_set_7seg_text(t, "START", 0x00);
tm1638_set_8leds(t, 0, 0);
delay(1000);

while(x!=S6)
{
      x = tm1638_read_8buttons(t);
      char text[10];
      FILE * proc;

      gps_detector();
      if(atoi(gps)!=0) tm1638_set_led(t, 0, 2); // ( LOAD PINS, LED POS,
COLOR )
      else tm1638_set_led(t, 0, 1);             // LED 0
      if(x==S3){
            delay(10);
            tm1638_set_7seg_text(t, "SCAN", 0x38);
            tm1638_set_led(t, 3, 2);
            wifi_scanner(SAMPLES);
            if(wi_ret) tm1638_set_7seg_text(t, "---", 0x00);
            *gpslatlon = gps_data();
            if(flag==0){
            lt1 = gpslatlon[0];
            ln1 = gpslatlon[1];
            gpslatlon[2] = gpslatlon[0];
            gpslatlon[3] = gpslatlon[1];
            flag=1;
            }
            lt1 = gpslatlon[2];
            ln1 = gpslatlon[3];
            lt2 = gpslatlon[0];
```

```
            ln2 = gpslatlon[1];
            distance = calcDistance( lt1, ln1, lt2, ln2);
            printf("\nReference Latitude: %.9f  and Longitude: %.9f",
lt1, ln1);
            printf("\nNew Latitude: %.9f  Longitude: %.9f", lt2, ln2);
            printf("\nDistance From Reference Point: %.0f\n", distance);
            make_csv();
            snprintf(text, 9, "%.0f %.0f", average, distance);
            tm1638_set_7seg_text(t, text, 0x00);
            tm1638_set_led(t, 3, 0);
      }
      kismet_server_detector();
        if(atoi(kismets)!=0) tm1638_set_led(t, 1, 2); //LED 1
        else tm1638_set_led(t, 1, 1);
        kismet_client_detector();
        if(atoi(kismetc)!=0) tm1638_set_led(t, 2, 2);    //LED 2
        else tm1638_set_led(t, 2, 1);
      if(x==S1){
            delay(10);
            tm1638_set_7seg_text(t, "KIS ON", 0x00);
            kismet_start();
      }
      if(x==S2){
            delay(10);
            tm1638_set_7seg_text(t, "KIS OFF", 0x00);
            kismet_stop();
      }
        if(x==S7){
            delay(10);
            tm1638_set_7seg_text(t, "reboot", 0x00);
            tm1638_set_8leds(t, 255, 0);
                rpi_reboot();
        }
        if(x==S8){
            delay(10);
                tm1638_set_7seg_text(t, "", 0x00);
            tm1638_set_8leds(t, 0, 0);
            tm1638_free(&t);
                rpi_shutdown();
        }
}
tm1638_set_7seg_text(t, "", 0x00);
tm1638_set_8leds(t, 0, 0);
tm1638_free(&t);
return 0;
}

int rpi_reboot()
{
FILE * process;
char rpi[10];

process = popen("sudo reboot", "r");
memset(rpi,'\0',sizeof(rpi));
rpi[lastchar] = '\0';
lastchar = fread(rpi, 1, 10, process);
pclose(process);
return 0;
}

int rpi_shutdown()
```

```
{
FILE * process;
char shut[10];

process = popen("sudo halt", "r");
memset(shut,'\0',sizeof(shut));
shut[lastchar] = '\0';
lastchar = fread(shut, 1, 10, process);
pclose(process);
return 0;
}

int gps_detector()
{
FILE * process;

process = popen("pgrep gpsd", "r");
memset(gps,'\0',sizeof(gps));
gps[lastchar] = '\0';
lastchar = fread(gps, 1, 10, process);
pclose(process);
return 0;
}

int kismet_server_detector()
{
FILE * process;

process = popen("pgrep kismet_server", "r");
memset(kismets,'\0',sizeof(kismets));
kismets[lastchar] = '\0';
lastchar = fread(kismets, 1, 10, process);
pclose(process);
return 0;
}

int kismet_client_detector()
{
FILE * process;

process = popen("pgrep kismet_client", "r");
memset(kismetc,'\0',sizeof(kismetc));
kismetc[lastchar] = '\0';
lastchar = fread(kismetc, 1, 10, process);
pclose(process);
return 0;
}

int kismet_start()
{
FILE * process;
char kismet_s[10];
char kismet_c[10];

process = popen(START_KISMET, "r");
memset(kismet_s,'\0',sizeof(kismet_s));
kismet_s[lastchar] = '\0';
lastchar = fread(kismet_s, 1, 10, process);

process = popen(START_KISMET_CLIENT, "r");
memset(kismet_c,'\0',sizeof(kismet_c));
```

```
kismet_c[lastchar] = '\0';
lastchar = fread(kismet_c, 1, 10, process);
return 0;
}

int kismet_stop()
{
FILE * process;
char kismetstop[10];

process = popen(STOP_KISMET, "r");
memset(kismetstop,'\0',sizeof(kismetstop));
kismetstop[lastchar] = '\0';
lastchar = fread(kismetstop, 1, 10, process);
pclose(process);
return 0;
}

int wifi_scanner(int samples)
{
FILE * process;
FILE * fp;
time_t ltime; /* calendar time */
ltime=time(NULL); /* get current cal time */
char wiscan[80];
float sum=0;
float sum1=0;
float iwiscan_vec[samples];
memset(iwiscan_vec,'\0',sizeof(iwiscan_vec));
fp = fopen(WIFI_SCAN_LOGFILE_PATH, "a+");     /* read write and append */
fprintf(fp, "\nThe instant measures are:");
for(int i=0; i < samples;i++){
process = popen("sudo iwlist wlan0 scan|awk
'/level/{a=$3}/ESSID:\"mesh\"/{print a}'", "r");
memset(wiscan,'\0',sizeof(wiscan));
wiscan[lastchar] = '\0';
lastchar = fread(wiscan, 1, 80, process);
strncpy(wiscan, wiscan+6, 5);
iwiscan_vec[i] = atoi(wiscan);
fprintf(fp, " %.0f ,", iwiscan_vec[i]);
}
for (int i = 0; i < samples; i++)
{
sum = sum + iwiscan_vec[i];
average = sum / samples;
}
for (int i = 0; i < samples; i++)
{
sum1 = sum1 + pow((iwiscan_vec[i] - average), 2);
}
variance = sum1 / (samples-1);
std_dev = sqrt(variance);

if(variance > 1){
fprintf(fp, "\nMesh Signal Level: %.0f dBm , Std Deviation: %.2f ,
Variance: %.2f , Time: %s", average, std_dev, variance, asctime(
localtime(&ltime) ) );
wi_ret=0;
}
else{
```

```
printf("\nThis sample was discarded and it's not going to be logged.
variance < 1\n");
printf("Take another sample\n");
wi_ret=1;
}
fclose(fp);
pclose(process);
return 0;
}

float calcDistance(float lat1, float lon1, float lat2, float lon2)
{
float dLat;
float dLon;
dLat = (lat2 - lat1);
dLon = (lon2 - lon1);
dLat /= 57.29577951;
dLon /= 57.29577951;
float v_a;
float v_c;
float distance;
float r = 6371;
v_a = sin(dLat/2) * sin(dLat/2) + cos(lat1) * cos(lat2) * sin(dLon/2) *
sin(dLon/2);
v_c = 2 * atan2(sqrt(v_a),sqrt(1-v_a));
distance = r * v_c;
distance = distance * 1000;
return distance;
}

float gps_data()
{
FILE * process;
float lat;
float lon;
char gps_lat[15];
char gps_lon[15];

do{
process = popen("gpspipe -w -n 5 | awk '/\"lat\":/{print $1}' | cut -d: -
f10-11", "r");
memset(gpsdata,'\0',sizeof(gpsdata));
memset(gps_lat,'\0',sizeof(gps_lat));
memset(gps_lon,'\0',sizeof(gps_lon));
gpsdata[lastchar] = '\0';
gps_lat[lastchar] = '\0';
gps_lon[lastchar] = '\0';
lastchar = fread(gpsdata, 1, 80, process);
strncpy(gps_lat, gpsdata, 12);
strncpy(gps_lon, &gpsdata[19], 12);
gpslatlon[0] = atof(gps_lat);
gpslatlon[1] = atof(gps_lon);
}while(gpslatlon[0] == 0 || gpslatlon[1] == 0);
pclose(process);
return *gpslatlon;
}

void make_csv()
{
time_t ltime; /* calendar time */
ltime=time(NULL); /* get current cal time */
```

```
FILE * fp;
fp = fopen(CSV_FILE_PATH, "a+");      /* read write and append */
if(flag_csv == 0)fprintf(fp, "RSSI (dBm),Std Deviation,Variance,Reference
Latitude,Reference Longitude,Latitude,Longitude,Distance (m),Time\n");
flag_csv=1;
fprintf(fp, "%.0f,%.2f,%.2f,%.9f,%.9f,%.9f,%.9f,%.0f,%s", average,
std_dev, variance, lt1, ln1, lt2, ln2, distance, asctime(
localtime(&ltime) ) );
fclose(fp);
}
```

Appendix E

In this appendix is the source code of the “shark” script written in bash.

```
#!/bin/sh
if [ $# -eq 0 ]; then
        echo " "
        echo "By default tshark will start in interface 4, running for
120 seconds."
        echo "For other options please use: ./shark.sh help"
        sleep 3
        printf "\nTshark Started at: %s $(date -u)"
        printf "\nRunning tshark for: 120 seconds."
        tshark -i 4 -a duration:120 -w tshark"$(date
+"%Y%m%d%H%M%S").pcap"
        printf "\nEnded at: %s $(date -u)"
        printf "\nThe log file name is: %s tshark$(date
+"%Y%m%d%H%M%S").pcap"
elif [ "$1" = "help" ]; then
        echo " "
        echo "Please choose whether you want to limit tshark by '"time"',
or '"packets"' and set it on the first argument."
        echo "Please choose the interface below and set correspondent
number in the second argument."
        tshark -D
        echo "Please set the (duration in seconds) or number of packets
in the third argument."
        printf "\nThe log file name will be: %s tshark$(date
+"%Y%m%d%H%M%S").pcap"
        echo "EXAMPLE TIME: ./shark.sh time 2 60"
        echo "EXAMPLE PACKETS: ./shark packets 2 20"
elif [ "$1" = "time" ]; then
        printf "\nTshark Started at: %s $(date -u)"
        printf "\nRunning tshark for: $3 seconds."
        tshark -i "$2" -a duration:"$3" -w tshark"$(date
+"%Y%m%d%H%M%S").pcap"
        printf "\nEnded at: %s $(date -u)"
        printf "\nThe log file name is: %s tshark$(date
+"%Y%m%d%H%M%S").pcap"
elif [ "$1" = "packets" ]; then
        printf "\nTshark Started at: %s $(date -u)"
        printf "\nRunning tshark for: $3 packets."
        tshark -i "$2" -c "$3" -w tshark"$(date +"%Y%m%d%H%M%S").pcap"
        printf "\nEnded at: %s $(date -u)"
        printf "\nThe log file name is: %s tshark$(date
+"%Y%m%d%H%M%S").pcap"
else
        echo " "
        echo "Only '"time"' or '"packets"'. are accepted as a valid
argument"
fi
exit 0
```

Appendix F

The code presented in this appendix is the TCP Server Application developed for the On Board Units (OBUs). This Application was written in “C” language and is composed by 3 source code files: the `tcpServerMIPS.c`, the `accelerometer.c` and the `gps.c`.

The following code is from the `tcpServerMIPS.c`, file.

```
/**
 * tcpServerMIPS.c
 * Author: Bruno Fernandes **/
#include "sensors.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netinet/in.h>

#define PORT 4444
#define MAX_CONN 5
#define GET 1
#define GPS 2
#define ACCEL 3
#define EXIT 99
#define RES_OK 0
#define ERR_GEN_FAIL -1

/* FUNCTION PROTOTYPES */
int compare(char command[], char arguments[]);

int main(int argc, char** argv) {

 struct sockaddr_in addr;
 struct sockaddr_in cl_addr;
 int sockfd, ret, newsockfd;
 socklen_t len;
 char *buffer;
 pid_t childpid;
 char clientAddr[INET_ADDRSTRLEN];
 int iCommand = 0;
 char *pStr = NULL;
 int iSize = 0;

 buffer = (char*)malloc(1024);
 sockfd = socket(AF_INET, SOCK_STREAM, 0);
 if (sockfd < 0) {
  printf("Error creating socket!\n");
  return -10;
 }
 printf("Socket created...\n");
```

```
 memset(&addr, '\0', sizeof(addr));
 addr.sin_family = AF_INET;
 addr.sin_addr.s_addr = INADDR_ANY;
 addr.sin_port = PORT;

 ret = bind(sockfd, (struct sockaddr *) &addr, sizeof(addr));
 if (ret < 0) {
  printf("Error binding!\n");
  return -20;
 }
 printf("Binding done...\n");
 printf("Waiting for a connection...\n");
 listen(sockfd, MAX_CONN);

 while(1) {
  len = sizeof(cl_addr);
  newsockfd = accept(sockfd, (struct sockaddr *) &cl_addr, &len);
  if (newsockfd < 0) {
   printf("Error accepting connection!\n");
   return -30;
  }
  printf("Connection accepted...\n");
  inet_ntop(AF_INET, &(cl_addr.sin_addr), clientAddr, INET_ADDRSTRLEN);
  if ((childpid = fork()) == 0) {
  close(sockfd);
    /*stop listening for new connections by the main process.
     *the child will continue to listen.
     *the main process now handles the connected client.
    */
   while(1) {
    memset (buffer, '\0', 1024);
    ret = recv(newsockfd, buffer, 1024, 0);
    if(ret < 0) {
     printf("Error receiving data!\n");
     return -40;
    }
    if(compare("GET", buffer))
        iCommand = GET;
    else if(compare("GPS", buffer)){
        iCommand = GPS;
    }
    else if(compare("ACCEL", buffer))
        iCommand = ACCEL;
    else if(compare("EXIT", buffer))
        iCommand = EXIT;
    else iCommand = 98;
    memset (buffer, '\0', 1024);

    switch(iCommand)
    {
        case GET:
            {
            char *pGps = NULL, *pAcc = NULL;
            char *pTotal = buffer; /* pTotal points to the beggining of
the buffer to send */
            int iSizeGps = 0, iSizeAcc = 0;
            if(gps(&pGps, &iSizeGps) != RES_OK) {
                printf("\nGPS gave out an error\n");
                break;
            }
            if(accelerometer(&pAcc, &iSizeAcc) != RES_OK) {
```

```
                printf("\nAccelerometer gave out an error\n");
                free(pGps);
                break;
            }
            if((iSizeGps + iSizeAcc) <= 1024) {
                memcpy(pTotal, pGps, iSizeGps); /* Copy the values from
the gps */
                memcpy(pTotal + iSizeGps, pAcc, iSizeAcc); /* Copy the
accelerometer values and adding the offset of the pTotal. */
            }
            else {
                printf("\nI need more memory on the buffer :(\n");
                free(pGps);
                free(pAcc);
                break;
            }
                printf("\n%s\n",pTotal);
                free(pGps);
                free(pAcc);
            }
                break;
        case GPS:
            if(gps(&pStr, &iSize) != RES_OK) {
                printf("\nGPS gave out an error\n");
                break;
            }
            if(iSize <= 1024)
                memcpy(buffer, pStr, iSize);
                printf("\n%s\n",pStr);
                free(pStr);
            break;
        case ACCEL:
            if(accelerometer(&pStr, &iSize) != RES_OK) {
                printf("\nAccelerometer gave out an error\n");
                break;
            }
            if(iSize <= 1024)
                memcpy(buffer, pStr, iSize);
                printf("\n%s\n",pStr);
                free(pStr);
            break;
        case EXIT:
            memcpy(buffer, "EXIT\n", strlen("EXIT\n"));
            exit(EXIT_SUCCESS);
            break;
        default:
            memcpy(buffer, "LIXO\n", strlen("LIXO\n"));
            break;
    }
    printf("Received data from %s: %s\n", clientAddr, buffer);
    ret = send(newsockfd, buffer, 1024, 0);
    if (ret < 0) {
     printf("Error sending data!\n");
     return -50;
    }
    printf("Sent data to %s: %s\n", clientAddr, buffer);
   }
  free(buffer);
  }
  close(newsockfd);
 }
```

```
 return 0;
}

int compare(char command[], char arguments[])
{
   int c = 0;
   while( command[c] == arguments[c] )
   {
      if( command[c] == '\0' || arguments[c] == '\0' )
         break;
      c++;
   }
   if( command[c] == '\0' && (arguments[c] == '\0' || arguments[c] ==
'\n' || arguments[c] == '\r'))
      return 1;
   else
      return 0;
}
```

The following code is from the `accelerometer.c`, file.

```
/** accelerometer.c
 *  Author: Bruno Fernandes **/
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdint.h>
#include <errno.h>
#include <string.h>
#include <semaphore.h>
#include <unistd.h>

#define SHMOBJ_PATH           "/drivein-gpsinfo"
#define SHMSEM_NAME           "/drivein-gpsinfo-sem"
#define ACCEL_LOGFILE_PATH      "/root/bruno/tcpServerMIPS/accellog.txt"
#define RES_OK 0
#define ERR_GEN_FAIL -1
#define MAX_SATS    20

struct ShmData {
      uint64_t systime;
      uint32_t gpstime;

      float speed; /* Km/h */
      float acc_output[3];
      uint8_t pwmgmt_status; /* 1: stopped, 2: moving, 3: going down */
};
typedef struct ShmData ShmData;

int accelerometer(char **ppStr, int *pSize){

    ShmData *shmdata = 0;
    sem_t *shmsem;
    FILE *fp;
    char *pStr = malloc(1024);
    int shmfd = 0;

    fp = fopen(ACCEL_LOGFILE_PATH, "a+");      /* read write and append */
```

```
    if ((shmfd = shm_open(SHMOBJ_PATH, O_RDONLY, S_IRUSR | S_IRGRP)) <
0){
        perror("In shm_open(): ");
        exit(EXIT_FAILURE);
    }
    if ((shmdata = (ShmData *)mmap(NULL, sizeof(ShmData), PROT_READ,
MAP_SHARED, shmfd, 0)) == MAP_FAILED){
        perror("In mmap(): ");
        exit(EXIT_FAILURE);
    }
    /* The shared memory region is protected by a semaphore */
    if ((shmsem = sem_open(SHMSEM_NAME, 0, S_IRWXU | S_IRWXG, 1)) ==
SEM_FAILED) {
        perror("In sem_open(): ");
        exit(EXIT_FAILURE);
    }
    if (sem_wait(shmsem)){
        perror("In sem_wait(): ");
        exit(EXIT_FAILURE);
    }
    /* In case I want to print information to a file. */
      /*
            fprintf(fp, "Systime: %s\n",shmdata->systime);
            fprintf(fp, "Gpstime: %s\n",shmdata->gpstime);
            fprintf(fp, "Acelerometer X Axis: %s\n",shmdata-
>acc_output[0]);
            fprintf(fp, "Acelerometer Y Axis: %s\n",shmdata-
>acc_output[1]);
            fprintf(fp, "Acelerometer Z Axis: %s\n",shmdata-
>acc_output[2]);
            fprintf(fp, "Status: %d\n",(int) shmdata->pwmgmt_status);
            fprintf(fp, "Speed: %s\n",shmdata->speed);
            fprintf(fp, "\n");
      */
    if(pStr) {

sprintf(pStr,"INI\nSYST=%llu\nGPST=%u\nACCX=%d\nACCY=%d\nACCZ=%d\nSTAT=%d
\nSPED=%d\nEND\n", shmdata->systime, shmdata->gpstime, (int) shmdata-
>acc_output[0], (int) shmdata->acc_output[1], (int) shmdata-
>acc_output[2], (int) shmdata->pwmgmt_status, (int) shmdata->speed);
        *ppStr = pStr;
        *pSize = strlen(pStr);
        fclose(fp);
        sem_post(shmsem);
        return RES_OK;
    } else {  /* malloc didn't gave me memory!! */
        fclose(fp);
        return ERR_GEN_FAIL;
    }
    if (sem_post(shmsem)){
        perror("In sem_post(): ");
        exit(EXIT_FAILURE);
    }
 fclose(fp);
 exit(EXIT_SUCCESS);

}
```

The following code is from the `gps.c`, file.

```
/**
 * gps.c
 * Author: Bruno Fernandes
**/
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdint.h>
#include <errno.h>
#include <string.h>
#include <semaphore.h>
#include <unistd.h>

#define SHMOBJ_PATH           "/drivein-gpsinfo"
#define SHMSEM_NAME           "/drivein-gpsinfo-sem"
#define GPS_LOGFILE_PATH      "/root/bruno/tcpServerMIPS/gpslog.txt"
#define RES_OK 0
#define ERR_GEN_FAIL -1
#define MAX_SATS    20

struct ShmData {
      uint64_t systime;
      uint32_t gpstime;
      uint8_t fix;
      uint8_t nsats;
      float hdop;
      float lat;
      float lon;
      float alt; /* meters */
      float speed; /* Km/h */
      float head; /* Degrees to north */

      /* SkyView stuff */
      uint8_t visiblesats; /* Number of Satellites in view */
      uint8_t prn[MAX_SATS]; /* ID */
      uint16_t az[MAX_SATS]; /* Azimuth */
      uint8_t el[MAX_SATS]; /* Elevation */
      uint8_t snr[MAX_SATS]; /* Signal-Strength */
      uint8_t used[MAX_SATS]; /* If satellite was used to estimate fix */

      float acc_output[3];
      uint8_t pwmgmt_status;  /* 1: stopped, 2: moving, 3: going down */
};
typedef struct ShmData ShmData;

int gps(char **ppStr, int *pSize){

    ShmData *shmdata = 0;
    sem_t *shmsem;
    FILE *fp;
    char *pStr = malloc(1024);
    int shmfd = 0;

    fp = fopen(GPS_LOGFILE_PATH, "a+");     /* read write and append */

    if ((shmfd = shm_open(SHMOBJ_PATH, O_RDONLY, S_IRUSR | S_IRGRP)) <
0){
```

```
        perror("In shm_open(): ");
        exit(EXIT_FAILURE);
    }
    if ((shmdata = (ShmData *)mmap(NULL, sizeof(ShmData), PROT_READ,
MAP_SHARED, shmfd, 0)) == MAP_FAILED){
        perror("In mmap(): ");
        exit(EXIT_FAILURE);
    }
    /* The shared memory region is protected by a semaphore */
    if ((shmsem = sem_open(SHMSEM_NAME, 0, S_IRWXU | S_IRWXG, 1)) ==
SEM_FAILED) {
        perror("In sem_open(): ");
        exit(EXIT_FAILURE);
    }
    if (sem_wait(shmsem)){
        perror("In sem_wait(): ");
        exit(EXIT_FAILURE);
    }
      /* In case I want to print information to a file. */
      /*
            fprintf(fp, "Systime: %llu\n",shmdata->systime);
            fprintf(fp, "Gpstime: %u\n",shmdata->gpstime);
            fprintf(fp, "Fix: %d\n",(int) shmdata->fix);
            fprintf(fp, "Nsats: %d\n",(int) shmdata->nsats);
            fprintf(fp, "Hdop: %d\n",(int) shmdata->hdop);
            fprintf(fp, "Lat: %f\n",shmdata->lat);
            fprintf(fp, "Lon: %f\n",shmdata->lon);
            fprintf(fp, "Alt: %d\n",(int) shmdata->alt);
            fprintf(fp, "Speed: %d\n",(int) shmdata->speed);
            fprintf(fp, "Head: %d\n",(int) shmdata->head);
            fprintf(fp, "Status: %d\n",(int) shmdata->pwmgmt_status);
            fprintf(fp, "\n");
      */
        if(pStr) {

sprintf(pStr,"INI\nSYST=%llu\nGPST=%u\nGFIX=%u\nNSAT=%u\nHDOP=%d\nGLAT=%f
\nGLON=%f\nGALT=%f\nSPED=%d\nHEAD=%d\nSTAT=%d\nEND\n", shmdata->systime,
shmdata->gpstime, shmdata->fix, shmdata->nsats, (int) shmdata->hdop,
shmdata->lat, shmdata->lon, shmdata->alt, (int) shmdata->speed, (int)
shmdata->head, (int) shmdata->pwmgmt_status);
        *ppStr = pStr;
        *pSize = strlen(pStr);
        fclose(fp);
        sem_post(shmsem);
        return RES_OK;
    } else {  /* malloc didn't gave me memory!! */
        fclose(fp);
        return ERR_GEN_FAIL;
    }
    if (sem_post(shmsem)){
        perror("In sem_post(): ");
        exit(EXIT_FAILURE);
    }
 fclose(fp);
 exit(EXIT_SUCCESS);

}
```

Appendix G

This appendix contains the configurations in the `/etc/network/interfaces` for the 1st node (node101) of the Multi-hop Network.

Configuration of `/etc/network/interfaces` for OLSR protocol:

```
#LOOPBACK INTERFACE
auto lo
iface lo inet loopback

#INTERFACE CONFIGURATION FOR SSH
auto eth0
iface eth0 inet static
address 192.168.8.101
netmask 255.255.255.0

#INTERFACE CONFIGURATION FOR OLSR
allow-hotplug wlan0
iface wlan0 inet static
address 192.168.83.101
netmask 255.255.255.0
wireless-essid mesh
wireless-mode ad-hoc
wireless-channel 6

iface default inet dhcp
```

Configuration of `/etc/network/interfaces` for *Babel* protocol:

```
#LOOPBACK INTERFACE
auto lo
iface lo inet loopback

#INTERFACE CONFIGURATION FOR SSH
auto eth0
iface eth0 inet static
address 192.168.8.101
netmask 255.255.255.0

#INTERFACE CONFIGURATION FOR BABEL
allow-hotplug wlan0
iface wlan0 inet static
address 192.168.81.101
netmask 255.255.255.255
wireless-essid mesh
wireless-mode ad-hoc
wireless-channel 6

iface default inet dhcp
```

Appendix H

This appendix contains the configurations in the `/etc/network/interfaces` for the 2nd node (node102) of the Multi-hop Network.

Configuration of `/etc/network/interfaces` for OLSR protocol:

```
#LOOPBACK INTERFACE
auto lo
iface lo inet loopback

#INTERFACE CONFIGURATION FOR SSH
auto eth0
iface eth0 inet static
address 192.168.8.102
netmask 255.255.255.0

#INTERFACE CONFIGURATION FOR OLSR
allow-hotplug wlan0
iface wlan0 inet static
address 192.168.83.102
netmask 255.255.255.0
wireless-essid mesh
wireless-mode ad-hoc
wireless-channel 6

iface default inet dhcp
```

Configuration of `/etc/network/interfaces` for *Babel* protocol:

```
#LOOPBACK INTERFACE
auto lo
iface lo inet loopback

#INTERFACE CONFIGURATION FOR SSH
auto eth0
iface eth0 inet static
address 192.168.8.102
netmask 255.255.255.0

#INTERFACE CONFIGURATION FOR BABEL
allow-hotplug wlan0
iface wlan0 inet static
address 192.168.82.102
netmask 255.255.255.255
wireless-essid mesh
wireless-mode ad-hoc
wireless-channel 6

iface default inet dhcp
```

Appendix I

This appendix contains the configurations in the `/etc/network/interfaces` for the 3rd node (node103) of the Multi-hop Network.

Configuration of `/etc/network/interfaces` for OLSR protocol:

```
#LOOPBACK INTERFACE
auto lo
iface lo inet loopback
#INTERFACE CONFIGURATION FOR SSH
auto eth0
iface eth0 inet static
address 192.168.8.103
netmask 255.255.255.0
#INTERFACE CONFIGURATION FOR WLAN0 (STA)
auto wlan0
allow-hotplug wlan0
iface wlan0 inet dhcp
wireless-essid WiFi Porto Digital
#INTERFACE CONFIGURATION FOR OLSR
allow-hotplug wlan1
iface wlan1 inet static
address 192.168.83.103
netmask 255.255.255.0
wireless-essid mesh
wireless-mode ad-hoc
wireless-channel 6

iface default inet dhcp
```

Configuration of `/etc/network/interfaces` for *Babel* protocol:

```
#LOOPBACK INTERFACE
auto lo
iface lo inet loopback
#INTERFACE CONFIGURATION FOR SSH
auto eth0
iface eth0 inet static
address 192.168.8.103
netmask 255.255.255.0
#INTERFACE CONFIGURATION FOR WLAN0 (STA)
auto wlan0
allow-hotplug wlan0
iface wlan0 inet dhcp
wireless-essid WiFi Porto Digital
#INTERFACE CONFIGURATION FOR BABEL
allow-hotplug wlan1
iface wlan1 inet static
address 192.168.83.103
netmask 255.255.255.0
wireless-essid mesh
wireless-mode ad-hoc
wireless-channel 6

iface default inet dhcp
```

Appendix J

This appendix contains the scripts to configure the 1st node (node101) in the Multi-hop network.

The script `olsr.start` for node101.

```
#!/bin/sh
#This script starts OLSRd protocol Deamon with the following rules
#Configuring mesh interface
mesh_config() {
      echo " "
      echo "Configuring mesh interface wlan0"
      echo "NOTICE: wlan0 is being used for TL-WN722N WiFi dongle"
      echo "The USB WiFi dongle must support IBSS or Ad-Hoc mode"
      sudo ifconfig wlan0 down
      sudo iwconfig wlan0 mode ad-hoc channel 6 essid "mesh"
      sudo ifconfig wlan0 192.168.83.101 netmask 255.255.255.0 up
      sudo ifconfig wlan0 up
}
# Print the IP address
ip() {
        echo " "
        _IP=$(hostname -I) || true
        if [ "$_IP" ]; then
                printf "My IP address is %s\n" "$_IP"
        fi
}
#Checking if Protocol Daemons are running.
BABEL_PROC=`pgrep babeld`
OLSR_PROC=`pgrep olsrd`
echo " "
echo "Making sure no other protocol is running..."

if [ "${BABEL_PROC:-null}" = null ];
then
        echo "babel not running!"
else
        echo "babel is running!"
        sudo /etc/init.d/babeld stop
fi

if [ "${BABEL_PROC:-null}" != null ];
then
        echo "babel stoped!"
fi

if [ "${OLSR_PROC:-null}" = null ];
then
        #Configuring mesh interface
        mesh_config
        #Starting the Deamon
        echo " "
        echo "Starting OLSR Protocol Deamon..."
```

```
        sudo olsrd -d 0 -i wlan0 &
        ip
else
        echo "olsr is already running! Nothing to do here."
fi
echo " "
echo "All Done!"
exit 0
```

The script `olsr.stop` for node101.

```
#!/bin/sh
#This script stops OLSRd protocol Deamon with the following rules
#Checking if OLSR is already running. If not, Stops the Deamon.
OLSR_PROC=`pgrep olsrd`

if [ "${OLSR_PROC:-null}" = null ];
then
        echo "OLSR not running!"
else
        echo "OLSR is running!"
        echo "Stoping OLSR Protocol Deamon..."
        sudo killall olsrd
fi

if [ "${OLSR_PROC:-null}" != null ];
then
        echo " "
        echo "OLSR stoped Successfuly!"
fi
sudo ifconfig wlan0 down
echo " "
echo "All Done!"
exit 0
```

The script `babel.start` for node101.

```
#!/bin/sh
#This script starts BABELd protocol Deamon with the following rules
#Configuring mesh interface
mesh_config() {
      echo " "
      echo "Configuring mesh interface wlan0"
      echo "NOTICE: wlan0 is being used for TL-WN722N WiFi dongle"
      echo "The USB WiFi dongle must support IBSS or Ad-Hoc mode"
      sudo ifconfig wlan0 down
      sudo iwconfig wlan0 mode ad-hoc channel 6 essid "mesh"
      sudo ifconfig wlan0 192.168.81.101 netmask 255.255.255.255 up
      sudo ifconfig wlan0 up
}
# Print the IP address
ip() {
        echo " "
        _IP=$(hostname -I) || true
        if [ "$_IP" ]; then
                printf "My IP address is %s\n" "$_IP"
        fi
}
#Checking if Protocol Daemons are running.
BABEL_PROC=`pgrep babeld`
BABEL_FLAGS=`route -n | grep 192.168.83.103 | awk '{print $4}'`
```

```
OLSR_PROC=`pgrep olsrd`
echo " "
echo "Making sure no other protocol is running..."

if [ "${BABEL_PROC:-null}" = null ];
then
      #Configuring mesh interface
      mesh_config
      #Starting the Deamon
      echo " "
      echo "Starting BABEL Protocol Deamon..."
      sudo /etc/init.d/babeld start
      sleep 1
      sudo /etc/init.d/babeld stop
      sleep 1
      sudo /etc/init.d/babeld start
      ip
else
        echo "BABEL is already running! Nothing to do here."
fi

if [ "${BABEL_FLAGS}" = "!H" ];
then
        echo "Node103 rebooting... Babel has rejected routes !H"
        sudo reboot
fi

if [ "${OLSR_PROC:-null}" = null ];
then
        echo "OLSR not running!"
else
        echo "OLSR is running!"
        sudo killall olsrd
fi

if [ "${OLSR_PROC:-null}" != null ];
then
        echo "OLSR Stoped!"
fi
echo " "
echo "All Done!"
exit 0
```

The script `babel.stop` for node101.

```
#!/bin/sh
#This script stops BABELd protocol Deamon with the following rules
#Checking if BABEL is already running. If not, Stops the Deamon.
BABEL_PROC=`pgrep babeld`

if [ "${BABEL_PROC:-null}" = null ];
then
        echo "BABEL not running!"
else
        echo "BABEL is running!"
      echo "Stoping BABEL Protocol Deamon..."
      sudo /etc/init.d/babeld stop
fi

if [ "${BABEL_PROC:-null}" != null ];
then
```

```
        echo " "
        echo "BABEL Stopped Successfuly!"
fi
sudo ifconfig wlan0 down
echo " "
echo "All Done!"
exit 0
```

Appendix K

This appendix contains the scritps to configure the 2nd node (node102) in the Multi-hop network.

The script `olsr.start` for node102.

```
#!/bin/sh
#This script starts OLSRd protocol Deamon with the following rules
#Configuring mesh interface
mesh_config() {
      echo " "
      echo "Configuring mesh interface wlan0"
      echo "NOTICE: wlan0 is being used for TL-WN722N WiFi dongle"
      echo "The USB WiFi dongle must support IBSS or Ad-Hoc mode"
      sudo ifconfig wlan0 down
      sudo iwconfig wlan0 mode ad-hoc channel 6 essid "mesh"
      sudo ifconfig wlan0 192.168.83.102 netmask 255.255.255.0 up
      sudo ifconfig wlan0 up
}
# Print the IP address
ip() {
        echo " "
        _IP=$(hostname -I) || true
        if [ "$_IP" ]; then
                printf "My IP address is %s\n" "$_IP"
        fi
}
#Checking if Protocol Daemons are running.
BABEL_PROC=`pgrep babeld`
OLSR_PROC=`pgrep olsrd`
echo " "
echo "Making sure no other protocol is running..."

if [ "${BABEL_PROC:-null}" = null ];
then
        echo "babel not running!"
else
        echo "babel is running!"
        sudo /etc/init.d/babeld stop
fi

if [ "${BABEL_PROC:-null}" != null ];
then
        echo "babel stoped!"
fi

if [ "${OLSR_PROC:-null}" = null ];
then
        #Configuring mesh interface
        mesh_config
        #Starting the Deamon
        echo " "
        echo "Starting OLSR Protocol Deamon..."
```

```
        sudo olsrd -d 0 -i wlan0 &
        ip
else
        echo "olsr is already running! Nothing to do here."
fi
echo " "
echo "All Done!"
exit 0
```

The script `olsr.stop` for node102.

```
#!/bin/sh
#This script stops OLSRd protocol Deamon with the following rules
#Checking if OLSR is already running. If not, Stops the Deamon.
OLSR_PROC=`pgrep olsrd`

if [ "${OLSR_PROC:-null}" = null ];
then
        echo "OLSR not running!"
else
        echo "OLSR is running!"
        echo "Stoping OLSR Protocol Deamon..."
        sudo killall olsrd
fi

if [ "${OLSR_PROC:-null}" != null ];
then
        echo " "
        echo "OLSR stoped Successfuly!"
fi
sudo ifconfig wlan0 down
echo " "
echo "All Done!"
exit 0
```

The script `babel.start` for node102.

```
#!/bin/sh
#This script starts BABELd protocol Deamon with the following rules
#Configuring mesh interface
mesh_config() {
      echo " "
      echo "Configuring mesh interface wlan0"
      echo "NOTICE: wlan0 is being used for TL-WN722N WiFi dongle"
      echo "The USB WiFi dongle must support IBSS or Ad-Hoc mode"
      sudo ifconfig wlan0 down
      sudo iwconfig wlan0 mode ad-hoc channel 6 essid "mesh"
      sudo ifconfig wlan0 192.168.82.102 netmask 255.255.255.255 up
      sudo ifconfig wlan0 up
}
# Print the IP address
ip() {
        echo " "
        _IP=$(hostname -I) || true
        if [ "$_IP" ]; then
                printf "My IP address is %s\n" "$_IP"
        fi
}
#Checking if Protocol Daemons are running.
BABEL_PROC=`pgrep babeld`
BABEL_FLAGS=`route -n  | grep 192.168.83.103 | awk '{print $4}'`
```

```
OLSR_PROC=`pgrep olsrd`
echo " "
echo "Making sure no other protocol is running..."

if [ "${BABEL_PROC:-null}" = null ];
then
      #Configuring mesh interface
      mesh_config
      #Starting the Deamon
      echo " "
      echo "Starting BABEL Protocol Deamon..."
      sudo /etc/init.d/babeld start
      sleep 1
      sudo /etc/init.d/babeld stop
      sleep 1
      sudo /etc/init.d/babeld start
      ip
else
        echo "BABEL is already running! Nothing to do here."
fi

if [ "${BABEL_FLAGS}" = "!H" ];
then
        echo "Node103 rebooting... Babel has rejected routes !H"
        sudo reboot
fi

if [ "${OLSR_PROC:-null}" = null ];
then
        echo "OLSR not running!"
else
        echo "OLSR is running!"
        sudo killall olsrd
fi

if [ "${OLSR_PROC:-null}" != null ];
then
        echo "OLSR Stoped!"
fi
echo " "
echo "All Done!"
exit 0
```

The script `babel.stop` for node102.

```
#!/bin/sh
#This script stops BABELd protocol Deamon with the following rules
#Checking if BABEL is already running. If not, Stops the Deamon.
BABEL_PROC=`pgrep babeld`
if [ "${BABEL_PROC:-null}" = null ];
then
        echo "BABEL not running!"
else
        echo "BABEL is running!"
      echo "Stoping BABEL Protocol Deamon..."
      sudo /etc/init.d/babeld stop
fi

if [ "${BABEL_PROC:-null}" != null ];
then
        echo " "
```

```
        echo "BABEL Stopped Successfuly!"
fi
sudo ifconfig wlan0 down
echo " "
echo "All Done!"
exit 0
```

Appendix L

This appendix contains the scritps to configure the 3rd node (node103) in the multi-hop network.

The script `olsr.start` for node103.

```
#!/bin/sh
#This script starts OLSRd protocol Deamon with the following rules
#Configuring mesh interface
mesh_config() {
      echo " "
      echo "Configuring mesh interface wlan1"
      echo "NOTICE: wlan1 is being used for TL-WN722N WiFi dongle"
      echo "The USB WiFi dongle must support IBSS or Ad-Hoc mode"
      sudo ifconfig wlan1 down
      sudo iwconfig wlan1 mode ad-hoc channel 6 essid "mesh"
      sudo ifconfig wlan1 192.168.83.103 netmask 255.255.255.0 up
      sudo ifconfig wlan1 up
}
# Print the IP address
ip() {
      echo " "
      _IP=$(hostname -I) || true
      if [ "$_IP" ]; then
            printf "My IP address is %s\n" "$_IP"
      fi
}
#Checking if Protocol Daemons are running.
BABEL_PROC=`pgrep babeld`
OLSR_PROC=`pgrep olsrd`
echo " "
echo "Making sure no other protocol is running..."

if [ "${BABEL_PROC:-null}" = null ];
then
        echo "babel not running!"
else
        echo "babel is running!"
        sudo /etc/init.d/babeld stop
fi

if [ "${BABEL_PROC:-null}" != null ];
then
        echo "babel stoped!"
fi

if [ "${OLSR_PROC:-null}" = null ];
then
      #Configuring mesh interface
      mesh_config
      #Starting the Deamon
      echo " "
      echo "Starting OLSR Protocol Deamon..."
```

```
      sudo olsrd -d 0 -i wlan1 &
      #sudo /etc/init.d/olsrd start
      #sudo /etc/init.d/olsrd stop
      #sudo /etc/init.d/olsrd start
      ip
else
        echo "olsr is already running! Nothing to do here."
fi
echo " "
echo "All Done!"
exit 0
```

The script `olsr.stop` for node103.

```
#!/bin/sh
#This script stops OLSRd protocol Deamon with the following rules
#Checking if OLSR is already running. If not, Stops the Deamon.
OLSR_PROC=`pgrep olsrd`
if [ "${OLSR_PROC:-null}" = null ];
then
        echo "OLSR not running!"
else
        echo "OLSR is running!"
      echo "Stoping OLSR Protocol Deamon..."
        sudo killall olsrd
fi

if [ "${OLSR_PROC:-null}" != null ];
then
      echo " "
        echo "OLSR stoped Successfuly!"
fi
sudo ifconfig wlan1 down
echo " "
echo "All Done!"
exit 0
```

The script `babel.start` for node103.

```
#!/bin/sh
#This script starts BABELd protocol Deamon with the following rules
#Configuring mesh interface
mesh_config() {
echo " "
echo "Configuring mesh interface wlan1"
echo "NOTICE: wlan1 is being used for TL-WN722N WiFi dongle"
echo "The USB WiFi dongle must support IBSS or Ad-Hoc mode"
sudo ifconfig wlan1 down
sudo iwconfig wlan1 mode ad-hoc channel 6 essid "mesh"
sudo ifconfig wlan1 192.168.83.103 netmask 255.255.255.0 up
sudo ifconfig wlan1 up
}
# Print the IP address
ip() {
        echo " "
        _IP=$(hostname -I) || true
        if [ "$_IP" ]; then
                printf "My IP address is %s\n" "$_IP"
        fi
}
#Checking if Protocol Daemons are running.
```

```
BABEL_PROC=`pgrep babeld`
BABEL_FLAGS=`route -n  | grep 192.168.82.102 | awk '{print $4}'`
OLSR_PROC=`pgrep olsrd`
echo " "
echo "Making sure no other protocol is running..."

if [ "${BABEL_PROC:-null}" = null ];
then
      #Configuring mesh interface
      mesh_config
      #Starting the Deamon
      echo " "
      echo "Starting BABEL Protocol Deamon..."
      sudo /etc/init.d/babeld start
      sleep 1
      sudo /etc/init.d/babeld stop
      sleep 1
      sudo /etc/init.d/babeld start
      ip
else
        echo "BABEL is already running! Nothing to do here."
fi

if [ "${BABEL_FLAGS}" = "!H" ];
then
      echo "Node103 rebooting... Babel has rejected routes !H"
      sudo reboot
fi

if [ "${OLSR_PROC:-null}" = null ];
then
        echo "OLSR not running!"
else
        echo "OLSR is running!"
        sudo killall olsrd
fi

if [ "${OLSR_PROC:-null}" != null ];
then
        echo "OLSR Stoped!"
fi
echo " "
echo "All Done!"
exit 0
```

The script `babel.stop` for node103.

```
#!/bin/sh
#This script stops BABELd protocol Deamon with the following rules
#Checking if BABEL is already running. If not, Stops the Deamon.
BABEL_PROC=`pgrep babeld`
if [ "${BABEL_PROC:-null}" = null ];
then
        echo "BABEL not running!"
else
        echo "BABEL is running!"
        echo "Stoping BABEL Protocol Deamon..."
        sudo /etc/init.d/babeld stop
fi

if [ "${BABEL_PROC:-null}" != null ];
```

```
then
        echo " "
        echo "BABEL Stopped Successfuly!"
fi
sudo ifconfig wlan1 down
echo " "
echo "All Done!"
exit 0
```

Appendix M

TCP Server Application response to the "GPS" message request from a telnet client.

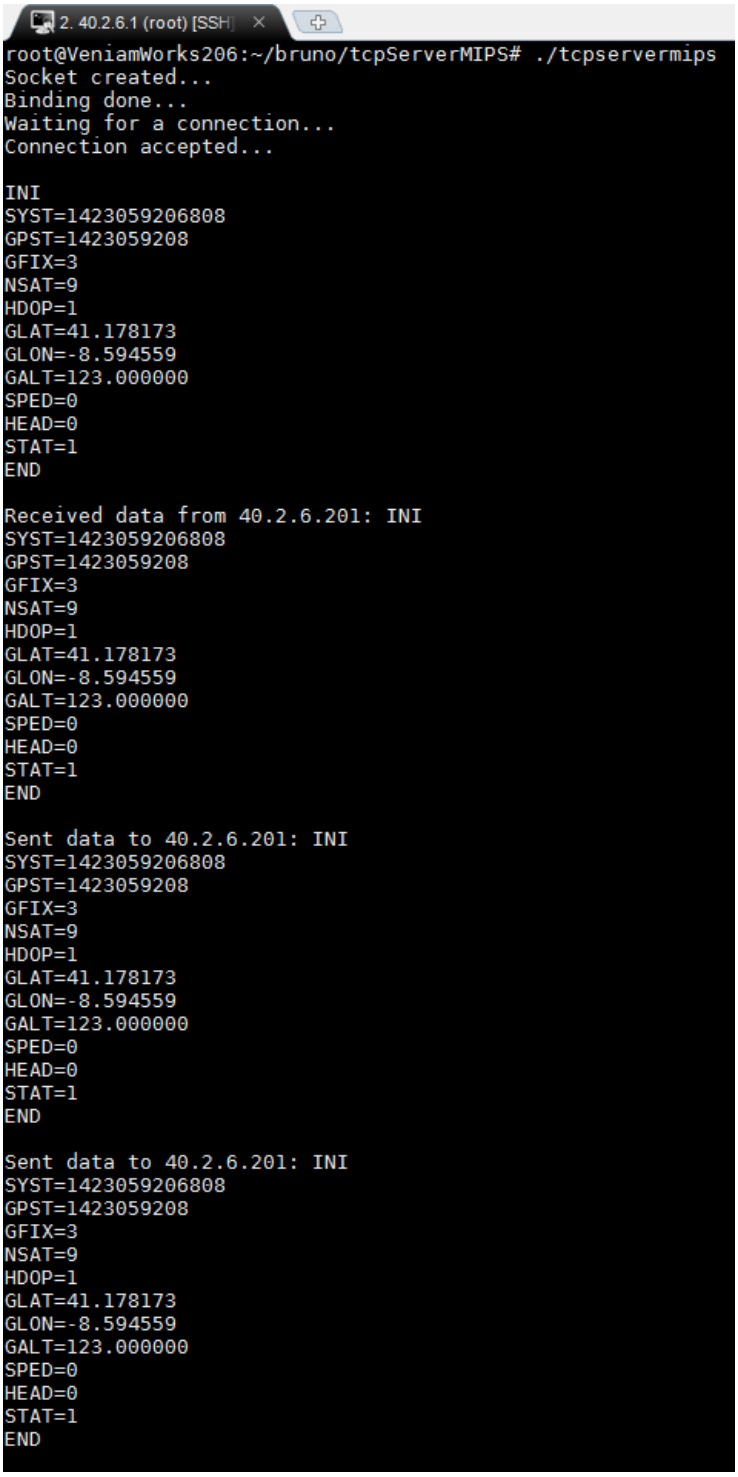

Figure 57 - TCP Server Application response to GPS message request

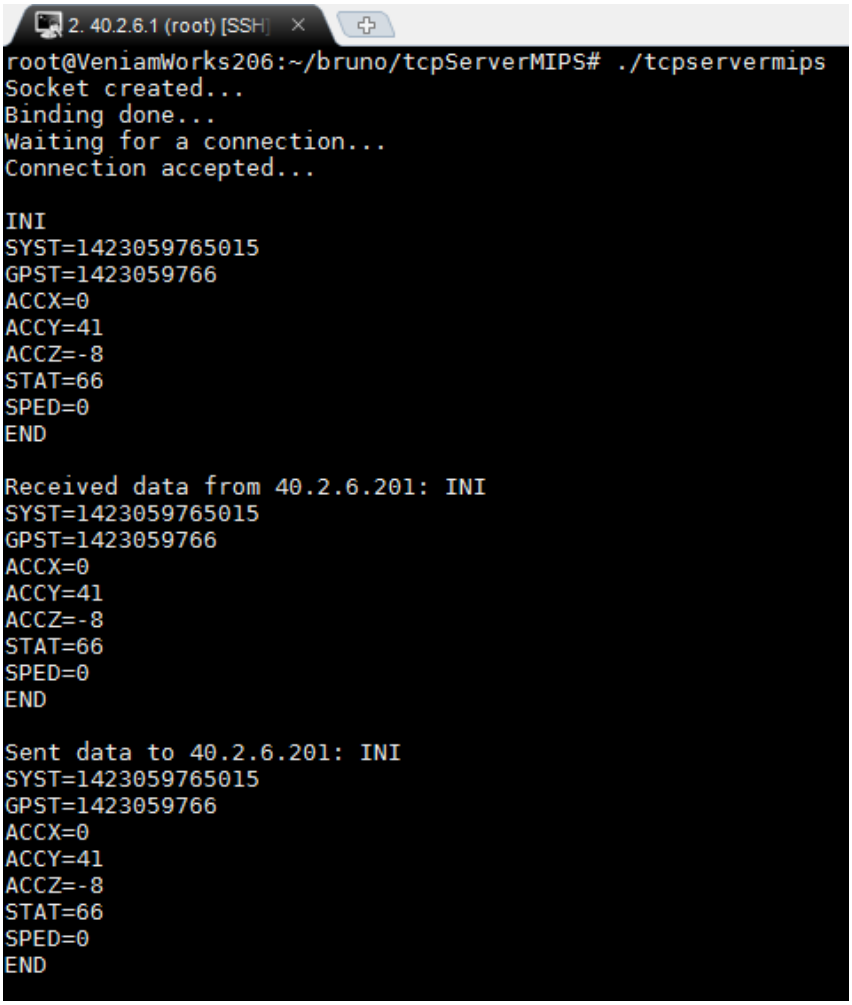

Figure 58 - TCP Server Application response to ACCEL message request

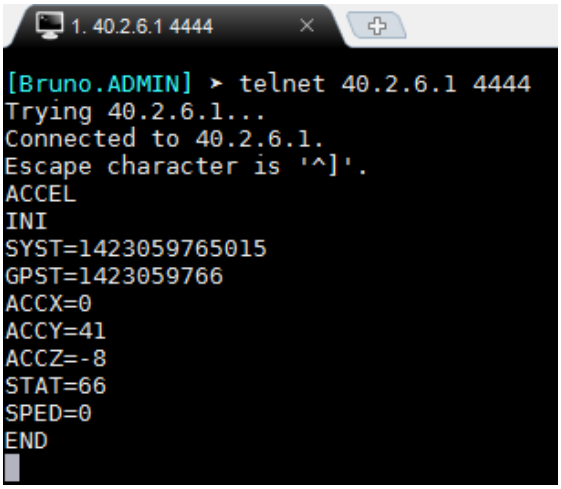

Figure 59 - OBU Telnet Client ACCEL message request / response

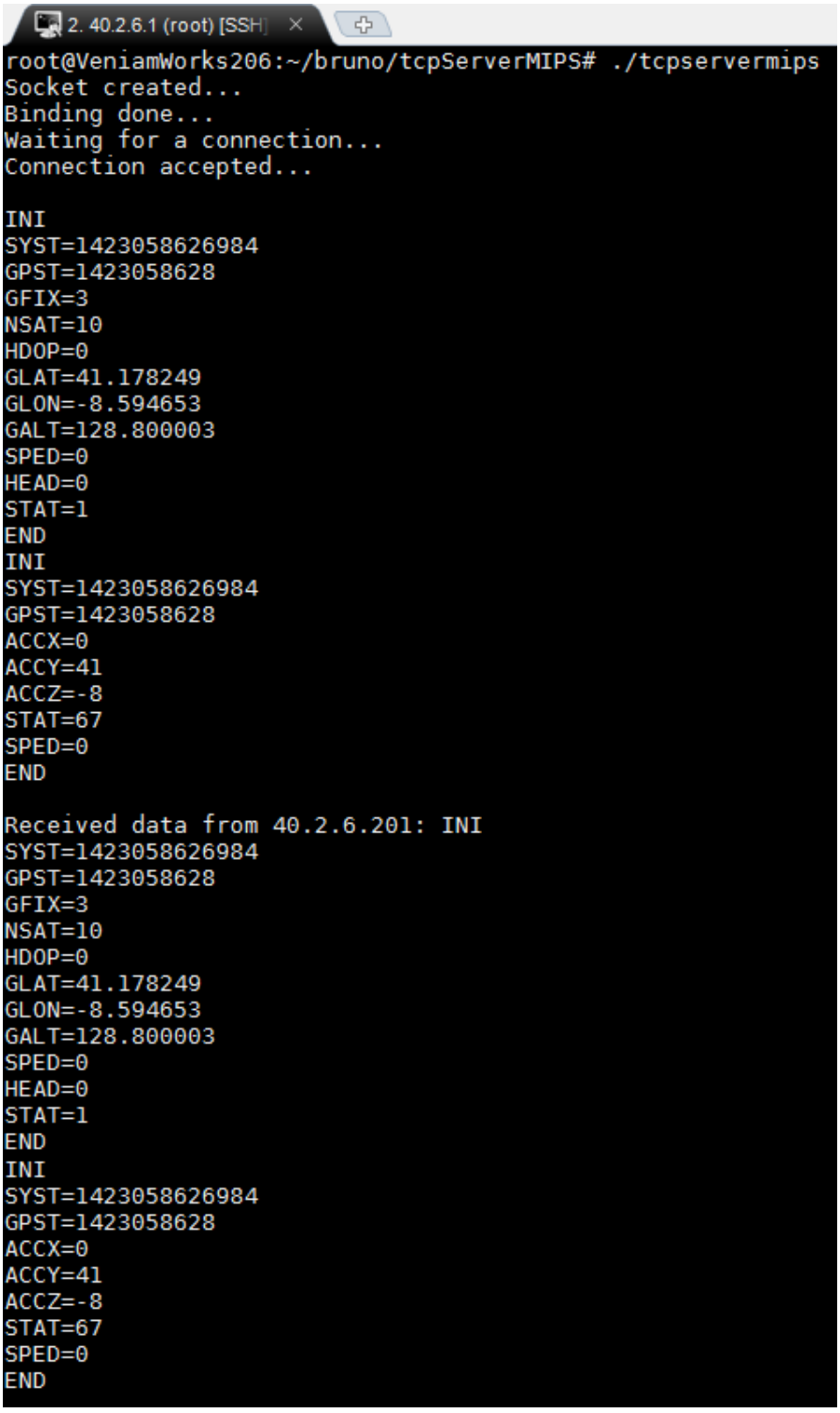

Figure 60 - TCP Server Application response to GET message request

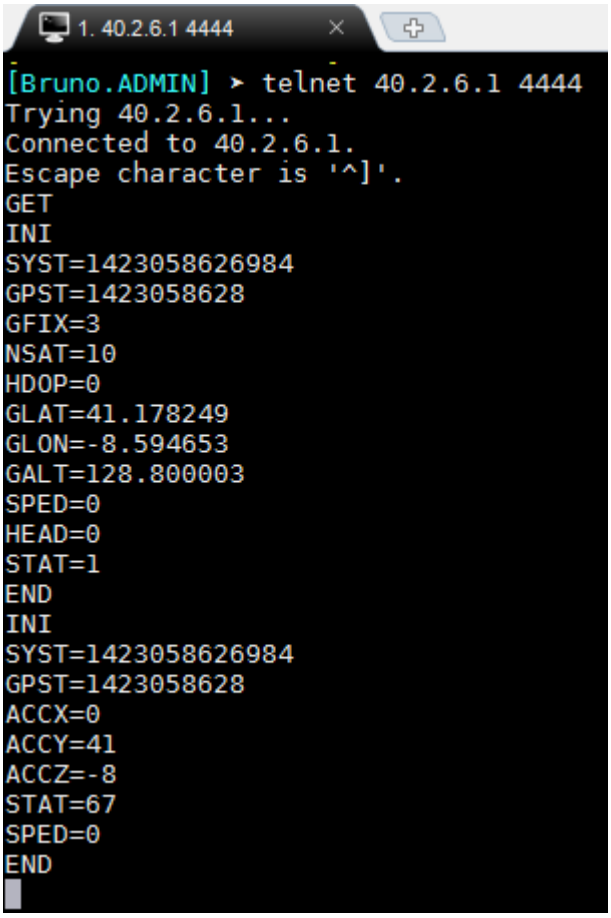

Figure 61 - OBU Telnet Client GET message request / response

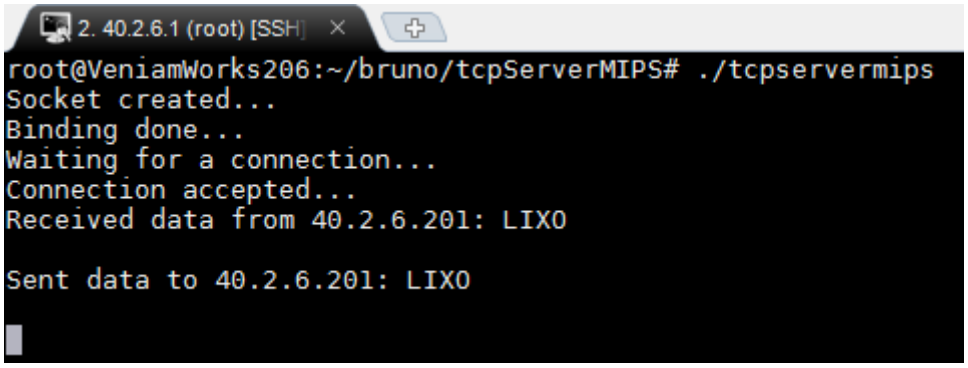

Figure 62 - TCP Server Application response to na invalid message request

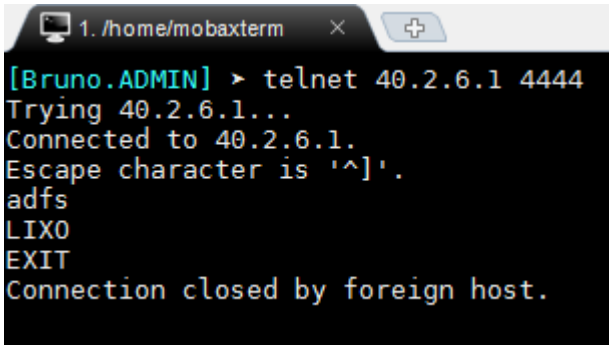

Figure 63 - OBU Telnet Client LIXO and EXIT message request / response

Appendix N

This appendix contains the script to start the Kismet [52], `gpsKismet.start` and the script to stop the Kismet [52], `gpsKismet.stop`.

```
#!/bin/sh
# Start GPS Daemon
sudo killall gpsd
echo "kill gpsd"
sudo gpsd /dev/ttyACM0 -F /var/run/gpsd.sock
echo "start gpsd"
sudo service ntp restart
echo "restart ntp"
# Set interface to monitor mode
sudo ifconfig wlan0 down
echo "wlan0 down"
sudo iwconfig wlan0 mode monitor
echo "monitor mode"
# Start Kismet Server
sudo kismet_server -s
#echo "kismet server started"
#echo "WARNING! This Script does not start kismet client"
#echo "Please start kismet client manualy ( sudo kismet_client -s )"
echo " "
echo "All Done!"
exit 0
```

```
#!/bin/sh
#This script stops Kismet Server and Client

KISMET_CLIENT_PROC=`pgrep kismet_client`
KISMET_SERVER_PROC=`pgrep kismet_server`

if [ "${KISMET_CLIENT_PROC:-null}" = null ];
then
        echo "Kismet Client is not running!"
else
        echo "Kismet Client is running!"
        echo "Stoping Kismet Client..."
        sudo killall kismet_client
fi
if [ "${KISMET_CLIENT_PROC:-null}" != null ];
then
        echo " "
        echo "Kismet Client stoped Successfuly!"
fi
if [ "${KISMET_SERVER_PROC:-null}" = null ];
then
        echo "Kismet Server is not running!"
else
        echo "Kismet Server is running!"
        echo "Stoping Kismet Server..."
        sudo killall kismet_server
```

```
fi
if [ "${KISMET_SERVER_PROC:-null}" != null ];
then
        echo " "
        echo "Kismet Server stoped Successfuly!"
fi
echo " "
echo "All Done!"
exit 0
```