BUILDING THE FIELD COMPONENT OF A SMART IRRIGATION SYSTEM: A DETAILED EXPERIENCE OF A COMPUTER SCIENCE GRADUATE

Submitted in fulfilment of the requirements for the degree of

MASTER OF SCIENCE

at Rhodes University

Yamnkelani Y. Pipile

March 2021

Abstract

South Africa is a semi-arid area with an average annual rainfall of approximately 450mm, 60 per cent of which goes towards irrigation. Current irrigation systems generally apply water in a uniform manner across a field, which is both inefficient and can kill the plants. The Internet of Things (IoT), an emerging technology involving the utilization of sensors and actuators to build complex feedback systems, present an opportunity to build a smart irrigation solution. This research project illustrates the development of the field components of a water monitoring system using off the shelf and inexpensive components, exploring at the same time how easy or difficult it would be for a general Computer Science graduate to use hardware components and associated tools within the IoT area. The problem was initially broken down through a classical top-down process, in order to identify the components such as micro-computers, micro-controllers, sensors and network connections, that would be needed to build the solution. I then selected the Raspberry Pi 3, the Arduino Arduino Uno, the MH-Sensor-Series hygrometer, the MQTT messaging protocol, and the ZigBee communication protocol as implemented in the XBee S2C. Once the components were identified, the work followed a bottom-up approach: I studied the components in isolation and relative to each other, through a structured series of experiments, with each experiment addressing a specific component and examining how easy was to use the component. While each experiment allowed the author to acquire and deepen her understanding of each component, and progressively built a more sophisticated prototype, towards the complete solution. I found the vast majority of the identified components and tools to be easy to use, well documented, and most importantly, mature for consumption by our target user, until I encountered the MQTT-SN (MQTT-Sensor Network) implementation, not as mature as the rest. This resulted in us designing and implementing a light-weight, general ZigBee/MQTT gateway, named "yoGa" (Yonella's Gateway) from the author. At the end of the research, I was able to build the field components of a smart irrigation system using the selected tools, including the yoGa gateway, proving practically that a Computer Science graduate from a South African University can become productive in the emerging IoT area.

Acknowledgements

Firstly, I would like to thank my supervisors Prof. Alfredo Terzoli and Dr Nomusa Dlodlo for their guidance and support. This successful completion of this research would not have been possible without you.

Secondly, I would like to thank my parents and siblings unwavering love and support. I would not be here without them.

Thirdly, I would like to extend my gratitude to my friends and colleagues for their continued support and encouragement.

Lastly, I would like to thank the Lord God, the for giving me the strength to complete this project.

This work was undertaken in the Distributed Multimedia Centre of Excellence at Rhodes University, with financial support from Telkom SA and INFINERA.

List of Tables

List of Figures

Listings

Contents

Chapter 1

Introduction

1.1 Background and Context

South Africa is a semi-arid country, the 30th driest country in the world [90]. The country receives an average annual rainfall less than 500 mm, which is less than half the Earth's average annual rainfall [90]. This average rainfall in unevenly distributed across different regions of the country, resulting in some areas being wetter than others, due to varying natural weather conditions [90]. South Africa has built over 500 governmental dams to store the water from the rainfall. These dams supply water in different regions of the country to ensure that communities do not run out of water in times of drought. In addition, the dams prevent flooding when there is an abundance of water [90].

1.2 The Role of Water in Agriculture

The water supplied to the communities is then used for various purposes, including domestic use, urban use, mining and industries, our main focus being the usage of water in the agricultural industry, specifically in plants and crops irrigation. Water plays an enormous role to all life, including the life of plants. Plants need water to grow and produce food, i.e, water is

needed to transport nutrients, which are needed by the plant to grow, from the soil to the plant leaves. In addition, water plays a significant role during photosynthesis. So, it is important to ensure that plants have the right amount of water, not too much or too little: over-watering suffocates the plants and possibly kills them. If the plants do not have enough water on the other hand, they will not receive nutrients from the soil, which also potentially kills the plants. The natural source of water for the plants is obviously rainwater. However, in dry season where there is not much rain, the plants need to be watered artificially through irrigation.

1.2.1 Internet of Things (IoT)

The Internet of Things is a network of devices with sensing and actuating capability, connected to the Internet and able to communicate and share information over a unified framework [98] [85][32]. Although the concept of IoT was first introduced by Kevin Ashton within scope of supply chain management, over the years IoTs have been applied across multiple domains [32]. For example, IoT has played a significant role in introducing the idea of smart homes and smart cities. IoT products have also been created in healthcare, educational and the agricultural sector [32]. IoT presents the next evolution to the Internet. Initially, the core functionality of the Internet was to connect computers. This function shifted towards connecting people. And now, IoT is primarily about connecting things. At the core of IoT are physical hardware elements with sensing and actuating capabilities which submit the collected data to the bigger Internet. These devices are attached to physical objects, allowing the transformation of these physical objects into "smart" objects that become aware of their environment: they can see, sense, communicate, interact and exchange data, knowledge and information [98]. IoT technologies can identify, locate, monitor these objects and trigger corresponding events autonomously in real-time [3]. The sensors are also often connected to programmable devices, such as micro-controllers, which are able to perform data collection from the sensing elements, and control the behavior of the actuating elements. Working with these hardware elements requires an understanding and knowledge of how to control sensors and actuators. Therefore, building IoT system requires an understanding in both computer science and electronic engineering.

1.3 Problem Statement

As I have stated earlier, water plays a significant role in plants. Because of the irregularity or scarcity of rain in many areas, artificial irrigation is often needed. Common irrigation techniques generally apply water to a field uniformly. However, different part of a field have different characteristics, such as soil composition or sun exposure. More and more sophisticated irrigation systems have been developed by commercial farmers to mitigate the problems due to this dis-homogeneity up to the most recent smart irrigation systems. The research reported in this thesis investigates two main, interrelated questions:

- What are the off-shelf, largely hobbyist and inexpensive components that can be used to build a smart irrigation solution?
- How easy would it be for a person with a Honours in Computer Science but no training in hardware, formal or informal, to use the needed components?

A third connected question is about the "maturity" of the various components, where the maturity of each component is measured by the amount of documentation and support available, in order to make it easy to use by non-specialists.

1.4 Research Goals

The primary goal of the work reported here is to investigate how easy or difficult it would be to build the field component of a water monitoring solution that controls the exact amount of water fed to a portion of a field based on the water content already in the soil. In order to achieve this goal, the following sub-goals must be realized:

- Run a scan of the domain in order to understand what has been done to create such a solution in order to identify components that can be used to build the solution.
- Select off shelf and inexpensive components that can be used to build the solution, through a top-down process.

- Use a bottom-up approach to learn and then integrate each component towards the final solution, recording the level of difficulty in each step.

1.5 Project Scope

The research project reported in this thesis is not to build a full smart irrigation solution, but the field component of such a system. The work includes IoT components to collect sensor data and actuate, which will clearly indicate the ease or difficulty of the hardware components for the primary researcher considering that hardware was not part of their standard education. As a result, there is no significant contribution on the back-end of the system, such as Machine Learning techniques and Artificial Intelligence, except for a simple database saver program reported during the implementation of the yoGa gateway (in Chapter 8) which served as an accessory component to illustrate the working of the gateway.

1.6 Methodology

The building of the system followed a constructive methodology in the sense that I actually construct the prototype of a full solution in the real world, examining all along the ease or difficulty of each step. The problem was initially broken down through a classical top-down process in order to identify the off-shelf and inexpensive components such as micro-controllers, sensors and network connections, that would be needed to build the solution. Once the components were identified, the work then followed a bottom-up approach where I started experimenting with the small components towards building the field component of a smart irrigation system. I studied the components in isolation and relative to each other. This was done through a structured series of experiments, with each experiment addressing a specific component and identifying how easy it is to use the component. This process progressively built a more sophisticated growing prototype towards the complete solution.

1.7 Document Structure

A diagrammatic representation of the structure and layout of this thesis can be seen in Figure 1.1 and is detailed in this section.

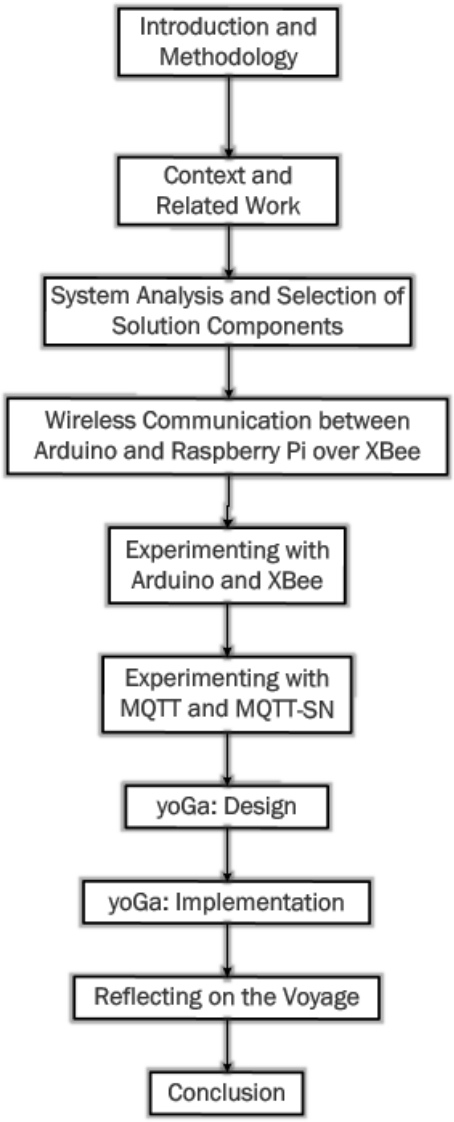

Figure 1.1: Thesis Structure

Chapter 2 - Context and Related Work: This chapter discusses the background and context of the problem, and explore existing related research work.

Chapter 3 - System Analysis and Selection of Solution Components: This chapter provides details on the top-down approach that I used to in order to analyse the problem, and identify and explore the off-the-shelf components that would be needed to build the field component of a smart irrigation system.

Chapter 4 - Experimenting with Arduino and XBee: This chapter reports on the experiments conducted with the Arduino micro-controller, including experiments to interface XBee modules with the Arduino micro-controller, and testing the wireless communication between XBee modules as well as Arduino micro-controllers.

Chapter 5 - Communicating Wirelessly between Raspberry Pi and Arduino: This chapter provides details on the experiments conducted with XBee devices to explore the use of these devices to enable a wireless communication between an Arduino micro-controller and a Raspberry Pi. In addition, this chapter explores and discusses the two supported ways in which XBee modules send and receive data, namely, the AT and API mode, drawing the differences between the two and stating the advantages that the API mode has over the AT operating mode.

Chapter 6 - MQTT and MQTT-SN: This chapter reports on the experiments conducted with the Raspberry Pi to explore MQTT messaging protocol, followed by an exploration of the MQTT-SN protocol, a super-set of the MQTT.

Chapter 7 - yoGa Gateway: Design: This chapter provides the analysis design of our own yoGa ZigBee/MQTT gateway in a classic top-down approach through a series of scenarios.

Chapter 8 - yoGa Gateway: Implementation: This chapter discusses the implemen-

tation of the yoGa gateway, with two accessory components to illustrate the working of the gateway in the context of a smart irrigation system.

Chapter 9 - Reflecting on the Voyage: In this chapter I present my experience and reflect on the ease or difficulty of each step towards the full system.

Chapter 10 - Conclusion: This chapter summarises the work done in this project and proposes ideas for future work.

Chapter 2

Context and Related Work

2.1 Introduction

This chapter aims at providing a description of the rich tapestry that represents the background of the work reported in this thesis. This chapter first explores the Fourth Industrial Revolution (4IR) and some of its technological underpinnings. Secondly, it provides a review of studies that indicate the adoption and usability of IoT projects in developing countries. The chapter then explores various IoT projects that have been developed for smart irrigation. This is followed by a discussion of the open source movement, and then the Maker Movement both important in this work, with a brief discussion of the Arduino and Raspberry Pi projects which are a part of both the open source and the Maker Movement.

2.2 Fourth Industrial Revolution (4IR) and the Internet of Things (IoT)

As defined by Klaus Schwab, the Founder and Executive Chairman of the World Economic Forum, the Fourth Industrial Revolution is characterized by technological advancements and innovations that break the barriers between the physical, digital, and biological worlds [73]. The revolution utilizes new technologies such as Artificial Intelligence (AI), Robotics, Cloud

Computing, Internet of Things (IoT), and 3D printing among others [73] [82] [46] [99].

2.2.1 4IR Technological Drivers

This sub-section expands on two technologies within the 4IR which are related to building a smart irrigation solution, the Internet of Things and Artificial Intelligence).

Internet of Things (IoT)

Digitization is the main driving force within 4IR. At the core of these digital technologies is the Internet of Things (IoT) [46]. IoT is defined as a network of objects devices with sensing and actuating capability. These devices are connected to the Internet and are able to communicate and share information over a unified framework [98] [85] [32]. Although the concept of IoT was first introduced by Kevin Ashton within the scope of the supply chain management. Over the years, IoTs have been applied across multiple domains [32]. For example, IoT has played a significant role in introducing the idea of smart homes and smart cities. IoT products are also seen in the healthcare, education and the agriculture sector [32]. IoT presents the next evolution to the Internet. Initially, the core functionality of the Internet was to connect computers. This function shifted towards connecting people. And now, the IoT is primarily about connecting "things". At the core of IoT are physical hardware elements with sensing and actuating capabilities, which submit the collected data to the bigger Internet. These devices are attached to physical objects and allow the transformation of these traditional physical objects into "smart" objects, which become aware of their environment. They can see, sense, communicate, interact and exchange data, knowledge and information [98]. IoT technologies can identify, locate, monitor these objects and trigger appropriate events autonomously in real-time [3]. The sensors are also often connected to programmable devices such as micro-controllers, which are able to perform data collection from the sensing elements, and control the behavior of the actuating elements. Working with these hardware elements therefore requires an understanding and knowledge of how to control sensors and actuators. Also, the elements are assembled in an electronic circuits and controlled by the micro-controller software.

Artificial Intelligence (AI)

Artificial Intelligence (AI) aims at mimicking and surpassing human intelligence using computers which are programmed to think like human beings, and also imitate their actions [46]. AI consists of numerous sub-fields including machine learning which is currently a major field. [46]. Machine learning uses algorithms to automatically learn, find hidden insights and improve from existing data without being programmed to do so [46]. Machine learning also enables these system to adapt and make repeatable and reliable decision when exposed to new data [46]. Machine learning has been widely deployed in image recognition, text-based sentiment analysis, bio-metric identification, etc. [72].

2.2.2 4IR Technologies in the Agricultural Sector

4IR offers potentially many opportunities, especially to developing countries, to create new forms of production that will trigger a period of valuable growth [10]. The use of sensors, big data, and machine learning could transform agricultural productivity everywhere and particularly Africa [60]. The effect of 4IR in agriculture is realized through the optimization of decision taking [82]. Agricultural decision taking involves the use of agricultural devices and equipment to collect data on soil conditions, as well as growth, climatic, and environmental information [1] [82] [101]. The accuracy does not only improve the farm productions but also adds to their value [21]. Information on competitive pricing, monitored crop information, disease prevention tips, and disaster mitigation support also poses an exceptional opportunity to remodel the agricultural sector to improve income, production, and demand throughout African countries [101] [82].

2.3 Aligning Developing Countries for IoT Adoption

As stated in Section 1.3, the main objective of this research work was to investigate how easy or difficult it would be for someone of my expertise in an upcoming country to interact with IoT components, and ultimately build a smart irrigation system. This section discusses existing

work that has been done in developing countries to align to the upcoming IoT technologies.

2.3.1 User Requirements for Internet Of Things (IoT) Applications: An Observational study

[56] conducted an observational study whose objective was to identify challenges encountered by users when they interact with IoT applications. The study objective was broken down into two main research questions. The first question focused on how the users were able to freely interact with the physical attributes of the IoT and without seeking assistance from the technical support team. The second question was to identify the challenges the users faced in monitoring the IoT application as it goes through self-adaptation. The study then identified and stated possible improvements to the IoT services and applications through user feedback. The study used a sample size of five participants with varying professional and technical profiles. The first user was a software engineer with no knowledge about IoT, the second was an experienced computer programmer who has a basic idea about what IoT is but has never used it for himself. The third and fourth participants were a restaurant owner and a nurser, respectively, neither had an understanding on what IoT is, and the latter had no knowledge even about technology in general. The fifth and last participant was a software developer with knowledge about IoT and its applications. The participants were handed a Smart Home starter kit, consisting of three devices: one that detected human presence or movements, open/close detector that indicated when the door was open or closed and a presence to indicate GPS location of an object once it was attached to the participants. The participants were then required to set up the application, placing the sensors on the correct positions object or person. The overall findings of the study indicated that the users ability to learn about how to use IoT depends solely on their background. Users with no experience with IT found the experience frustrating and overwhelming. In addition to the learnability of the applications, the users struggled to get started with setting up the elements of the application, they had no understanding of what to start with and they relied on "gut-feeling" and "try and error" methods.

Introduction of IoT Subjects to an Existing Curriculum, Israel

The Holon Institute of Technology has introduced an IoT elective course in their existing BSc program in Technology Management [59]. The course is structured to consist of both theoretical topics, which are taught in formal lectures and presentations, and hands-on practical training. The theoretical topics cover the history of IoT from M2M to IoT systems. They also cover the core IoT concepts, definitions, architecture, devices and applications in industry including a Service-oriented Architecture consisting of four layers (sending, network, service and interface), used for the design of many IoT systems. In the course practical, the students produce an IoT project plan and product development based on the IoT architecture using sensors, cloud computing, modules and software services. The project is implemented on Raspberry Pi computers using the Python programming language. During the practical, the students learn to install the Raspbian Linux operating system onto the Raspberry Pi. They also learn about the different GPIO pins on the Raspberry Pi, connecting different types of sensors (e.g moisture, temperature, light) to the Raspberry Pi, and interacting with the sensors (e.g performing measurements) using python. The students also learn installing a local database (SQLite) and using cloud databases such as Amazon Relational Database Service.

By the time the paper was written, the course had already been taught twice, to a total of 32 students in total. Most of these students had no prior experience in electronics and it was their first time working with hardware and hardware integration. However, during the course, it was discovered that the Raspberry Pi was a friendly single board computer, with which students could easily make IoT products. In addition, the Raspberry Pi website contains detailed instructions and information on how to work with the micro-computer. In the first instance of the course, the programming was taught in Java, which the students found difficult and complicated. As a result, they spent the majority of the time working out the language syntax, instead of on the actual project. In the second instance, the course programming language changed to Python. The students found working with Python easier, they were able to simultaneously learn the new language and focus on designing the application. It was concluded that the Python language is more suitable for people with no background in software engineering.

The future goals of the institution was to establish an advanced IoT laboratory to accelerate advanced projects both for research and educational purposes [59]. As a result, the laboratory will be equipped not only with the equipment that they already have (sensors, motors,lights, buttons, etc.), but also advanced components to build robots which will be implemented with the IoT applications. The institution also plans to update the existing "Introduction to Programming" course in the BSc program to teach Python, and also add a new course " Advanced topics in programming". In this way they hope to provide the students a better start for the IoT course [59].

Undergraduate Curriculum for Learning IoT in a Computer Science Faculty, Peru

The Computer Science faculty of the National University of San Marcos in Peru proposed an undergraduate curriculum for learning IoT topics to produce well trained graduates that can develop IoT applications meeting the needs of the Peruvian industry [33]. The proposed curriculum involved the introduction of new courses which cover the core topics relating to IoT. The first set of courses that the curriculum offers is a Digital Principles course, Computer Organization and a Control Engineering course, which provide topics about sensor networks. The curriculum also includes a Distributed Systems course which consists of topics about the middle-ware management layer. The curriculum is completed by a Software Engineering, and an Intelligent Systems course, where students learn about developing IoT applications and products [33].

During the alignment to new curriculum, the university implemented an IoT research lab [33]. This resulted a team of undergraduate students participating in the national science fair, "Science with Peru", where the students presented two IoT applications, one on monitoring the environment using Arduino and cloud data management [33]. The second project was about a brain/computer interface which controls a car for physically impaired persons. In addition, they managed to establish stronger relationships between the IoT research lab and various industries, where the companies fund some of the projects they develop, and hire se-

lected undergraduate students [33].

The IoT curriculum produced well-trained graduates with improved performance in developing novel applications involving IoT [33]. Also, the alignment also presented the students more opportunities to improve their professional career through the relationships they established between the IoT research lab and industries as I had mentioned earlier [33].

2.4 IoT Smart Irrigation Systems

2.4.1 Monitoring System Using Web of Things in Precision Agriculture

[43] proposed an alert system that controls the water stress of plants using IoT technology. The system was designed as a three tier application, consisting of the data acquisition tier, gateway tier, and the IoT tier. The data acquisition tier consists of a sensor network deployed in fields. Each sensor node in the network consist of an Atmega 128 micro-controller, IEEE 802.15.4 ZigBee Transceiver, SDRAM card and a soil moisture sensor. The sensor node is inserted in the soil to measure the water volume. The sensor nodes communicate using the 802.15.4 protocol, a protocol not directly supported in the Internet. A gateway then acts as a bridge between the 802.15.4 network and the GPRS network in order to allow connection between the sensor network and the Internet. The gateway is a Meshlium multi-protocol router, that supports 5 wireless connection interfaces, including ZigBee and 3G/GPRS [53]. Meshlium is also considered very user friendly for web applications. The Meshlium is hosted in a waterproof case, which makes it deployable outdoor. The gateway collects the sensor node data, and then stores it in a database, either locally or externally. The final tier of the application is the Cloud which allows users to interact with the system. The Ubidots platform was selected to capture sensor from the Meshlium gateway, and then publish the data to a cloud platform. Ubidots is a platform that enables developers to capture sensor data, and process it into meaningful information. The platform is also used to transmit the sensor data to an IoT cloud from an in-

ternet device. In addition, Ubidots allows developers to define alerts and triggers to automatic responses to preset threshold values. The overview in the user interface consists of a Google Maps that displays the farm field, with blue dots that indicate the location of each sensor node in the field. The application also indicates the soil moisture value associated with each sensor, and historical evolution of the moisture of each plot. In addition, the system sends an SMS alert when the water level exceeds a critical threshold value.

The display of the soil moisture status in each plot allows the farmers to estimate the amount of water required in a plot of land, which can improve the efficiency of water usage, and also increase crop yield. Furthermore, the historical information about the evolution of each plot soil moisture farmer information about the saturation or drought of each plot, which can help to decide which crop to plant in each plot, based on the moisture of the plot and the water requirements of the different plants. The notification when the water level reaches a critical point prevents water stress in the plants. However, the system can be improved into a more sophisticated one, for example, automate the irrigation so that the system is able to switch ON/OFF water supply in a plot based on its sensor readings, without requiring any human intervention. The system can also incorporate weather forecasting to improve the decision making.

2.4.2 A Low-Cost Smart Irrigation Control System

[71] proposed a prototype of a low cost, smart irrigation system that a middle class farmer can use in their farm field. The system consists of sensor nodes placed in different positions of a farm field, and a controller node. Each sensor node is equipped with a generic soil moisture sensor, and an Arduino Uno development board which consists of an ATMEGA-328 micro-controller. The soil moisture sensor interact with the soil and measure the soil moisture value, which is then transmitted to the micro-controller through a wireless networking device. Upon receiving this moisture value, the micro-controller calculates the percentage of the soil dryness using the following algorithm:

$$voltage = sensorvalue * (\frac{5}{1023})$$
$$percent = (\frac{voltage}{5}) * 100$$

The soil dryness percentage is then sent to the controller node, where it is further analyzed. The controller node is equipped with a Raspberry Pi micro-computer. The Raspberry Pi then controls the water motor valves in the different positions of the farm based on the percentage of dryness obtained from the sensor nodes. If the dryness or moisture exceeds the required values, then the water is open or closed off, respectively. In addition, the Raspberry connects with internet and transmits data such as the water motor status (ON/OFF) to the registered mobile number.

The experimental results of the prototype indicated that it is able to automatically control irrigation activities based on soil moisture readings. The irrigation of a field based on its water requirements reduces water wastage. In addition, the automatic irrigation reduces the need for human intervention. The authors note that the system devices consumes little power and so can be battery powered for a long time. The prototype was tested in a remote area, and was shown to be beneficial to farmers in such areas. In addition, the system is usable with all types of irrigation, eg. drip, channel, sprinkler.

2.4.3 Effective Implementation of Low-Cost Smart Irrigation System

[15] developed an automated and low cost smart irrigation solution, along with the ability to monitor weather parameters and providing security to a farm field. The system uses NodeMCU as a micro-controller, integrated with PH, soil moisture, and PIR sensors, and a WiFi module. Therefore acts as a WiFi module which may also act as a gateway through which the

system transmits data to the cloud. The NodeMCU also collects weather information such as temperature, humidity and clouds around the location of the plot, from an open weather api. This weather information is stored in the cloud. The PH sensor measures the ph value of the soil, which indicates the acidity, and hence suggesting to the farmer the type of crop to plant in the field. The soil moisture sensor monitors the dampness of the soil. The system uses the values from the sensor to switch ON/OFF water in the field when the value is below or exceeds a threshold value, respectively. The soil moisture data is also transmitted to the cloud where it is monitored through a web application, and may be retrieved in a spreadsheet format for future analysis. The PIR (Passive Infrared) sensor continuously monitors the field, and identifies predators (animals or human). When a predator is found, the farmer is notified through email.

The system was proved to be very low cost and consumes less power as it uses a NodeMCU as a micro-controller. In addition, the system is effective in the field of irrigation: the ability to monitor a field, without the need for the farmer to go physically is very useful. The ability to predict a crop for a field increases the crop yield and production. However, the PIR sensor has no indication of the type of object it detects, whether an animal or human, and this is a limitation of the system.

2.4.4 Smart Irrigation System

[75] designed an automated smart irrigation system which controls water flow based on the environmental conditions (temperature and humidity) in a plot using sensors. The system is equipped with an Arduino development board, integrated with a soil moisture and a temperature sensor which measure the soil humidity and the temperature in a plot. The micro-controller then switches ON/OFF the water flow in the plot based on these readings. In addition, the Arduino is connected to a GSM module, which enables the system to be connected to a smart phone. The system is then linked to a special android application, through which the farmer can see the workings of the system, including the temperature and humidity sensor values. The application also enables the farmers to manually control the irrigation activities, that is, switch ON/OFF the water valve based on the displayed sensor values. In addition smartphone, the

system irrigation activities can be remotely controlled from a designed web page available from www.sulamadenetim.com, also using the sensor values.

The use of the soil moisture sensor prevents unnecessary irrigation. This does not only save water, but also improves the crop yield. The ability to control the system with a smartphone makes it usable, as smartphones have a high usage rate nowadays. This usability is increased by the ability to control the irrigation remotely in a website.

2.4.5 IoT based Autonomous Percipient Irrigation System using Raspberry Pi

[38] proposed the design of an automatic water supplying system that is able to detect the appropriate time to water plants in a farmland, and continuously monitor the water level to prevent water accumulation around the rots of the saplings. The system consists of a central Raspberry Pi, Arduino micro-controller, various sensors, WiFi module, GSM shield, relay boards. Arduino micro-controllers placed in different positions in the farmland and collecting sensor data. The system uses a soil moisture sensor, daylight (or photocell) sensor, and a water level sensor. The soil moisture sensor interacts with the soil to measure the dryness of wetness of the soil, while the daylight sensor detects the sunbeam. In addition, a water level sensor was used to measure water level in the field. All the collected sensor data is transmitted to the Arduino micro-controller. Upon receiving the sensor data, the micro-controller wirelessly transmits the data to the central Raspberry Pi over WiFi. The Raspberry Pi keeps on traversing each Arduino by their IP listed on the Raspberry Pi, and requests sensor data from each Arduino. When the Raspberry Pi receives the sensor data from the Arduino, it compares them to preset soil moisture, photocell and water level threshold values. The Raspberry Pi then sends commands to the Arduino, to switch ON/OFF the water supply for a specified duration of time, and commands Arduino to switch off the water supply when the duration is exceeded. When the Arduino receives these commands, it simply takes action accordingly, open or close the water gate at the specific area. The system also consists of a water level sensor placed in the water tank, which detects the amount of water in the tank, and notify the administrator

when the water level is below a preset value. The Arduino uses the GSM shield equipped with a mobile SIM card to communicate with the administrator using SMS service.

The proposed system is scalable, it is usable in small pot plants, to a backyard garden, and to a bigger farmland. It also ensures a scientific and systematic approach to ensure that plants receive the right amount of water, thus improving crop production, and also ensuring a wiser water usage. The system could incorporate weather forecasting, which can help to improve the decision making and further avoid water wastage. The system can also be re-engineered to use an instrument that measures nutrients in the soil. Such an instrument could enable the system to supply fertilizers in the land in a precise manner.

2.4.6 Smart Irrigation System: A Water Management Procedure

[61] also designed a water management system to optimize water available on reservoirs in areas facing water scarcity issues in order to provide efficient and effective water usage. The system manages irrigation based on the soil moisture, and the amount of water available in a water reservoir. The system consists of a soil moisture sensor and an ultrasound sensor. The soil moisture sensor measures the amount of water in the soil, and the ultrasound sensor is placed on a reservoir, and then measures the water level. The values from the sensors are converted to an electronic signal which is then sent to the Atmega 328 micro-controller in Arduino Uno. Upon receiving the values, the micro-controller switches ON/OFF the water pumps when the values exceed preset threshold values. In addition, the system uses the water level value in the reservoir to prioritize areas to irrigate and determine the number of pumps to be activated at any instant.

The system ensures that plants are irrigated based on the soil water requirements, which in turn ensures plant growth growth and avoids water wastage. The ability to prioritize pump operations based on the the water level in the reservoirs guarantees longevity of irrigation pumps and also prevents wasting water. The system was tested in a laboratory with three different soil samples: dry, moist and wet. The results from the experiments indicated good

performance.

2.5 Open Source Movement

The open-source movement is a movement that support the idea of designing and implementing artefacts in a manner that makes them publicly available to access, modify and redistribute [95]. "Open source" means that other users can also share the design and implementation full details, and further improve or adapt the artefact. Common and successful open-source projects include the Linux operating system and the Apache project [44]. It also includes the Arduino development platform, which is widely used in the Internet of Things, and other movements such as the Maker Movement[44].

2.5.1 Open Source Software

Most software that is bought, or downloaded today normally comes in a compiled, ready-to-run version [95]. With this kind of software, the source code is only legally available to the person, team, or organisation that created it, and they are the only ones who can legally copy or alter the code, which then gives them full control over the software [95]. Open source software, on the other side totally rejects this idea of software source code being made available to a selected few [95]. Open source software supports the idea that software authors make a copy of the software source code available to users at no cost, and that the users can legally study the software code, modify it, and redistribute the modified version to the general public also at no cost [87]. Shared values and norms governed this idea of sharing software, sometimes referred to as software freedom norms, which maintain individual freedoms regarding the creation, modification, and sharing of software, explicitly granting users the following rights [66]. Firstly, users have the right to full access to the source code [66]. This full access to the source code allows users to gain an understanding of the inner workings of the software, enabling them to modify the software effectively as they deem appropriate. The modification of the source code can be anything from fixing a bug, or adding a feature [95]. Secondly, the users can then share both the original and the modified software for the benefit of the larger community [66].

Lastly, open source licensing forbids any form of discrimination against users, anyone can run the program for any purpose without any restriction [66].

2.5.2 Open Source Hardware

Open source hardware (or simply open hardware) refers to a physical artefact whose where all the tools and components that used to create the hardware design, are made publicly available so that anyone can access, modify or replicate the physical artefact [95]. For any hardware to qualify to be open source, the hardware must be release with its documentation (including design specifications and files) either attached to the physical object, or publicly published for easy download at a reasonable or no cost [94]. Also, users must be able to modify and redistribute these files [94]. In the case where some, and not all of the hardware design is released under an open source license, then the documentation should explicitly indicate which portion of the design is under the open source license [94]. Users must also be able to redistribute their modified versions of the file under the same license of the original work, and must also be granted the freedom to use the work for any purpose they desire [95].

2.5.3 Benefits of the Open Source Movement

The main idea of the open source movement is transparency between software and hardware authors and the users [66]. Users have full insight and understanding of the product code base, which makes open source hardware and software solutions reliable and secure, users know exactly what the code is doing, and the code is constantly under review by the community [95]. The ability for the users to modify hardware designs files and software code enables the users to build hardware and software solutions that meet their specific needs, i.e can examine the code, eliminate or even alter parts that they do not want, or even add functionality that they desire [4]. The idea of users sharing their modified versions allows them to share ideas among themselves, and instead of building solutions from scratch, they can build based on each other's solution, or even the the hardware original source code [95]. Open source code has also been used as an educational tool to learn building software [14].

2.5.4 Open Source Projects: Arduino

The Arduino project is an open-source, micro-controller-based development platform [29]. Its hardware is made up of a programmable micro-controller mounted on a circuit board, which provides easy access to the micro-controller input/output board and connectivity to a computer for programming and user interaction [29]. Arduino hardware is available under an open-source license called Creative Commons Attribution-Share Alike 2.5 [93] Not only the hardware components of Arduino are released under an open source licence, but also the software as well, including all its libraries and tools the LGPL and GPL open source license [93]. This makes the entire Arduino project to be under open source [7]. The platform is easy to use, with great support available online, and as a result, it has gained popularity in by hobbyist, artisans, and even the greater Maker Movement (I will talk about the movement in the following section). Arduino has also been used for educational purposes to prototype fun projects from simple to more advanced projects [7].

2.6 The Maker Movement

The Makers movement or Makers culture is a social movement that stands to support the doctrine that people can create, modify and re-purpose things themselves using traditional crafts or technology, instead of hiring a professional or buying them [64] [34] [25]. The ability to repair and re-purpose broken tools has always been crucial for human survival [34]. For years, people have been remodeling their homes, fixing leaking water taps, and decorating their clothes without consulting plumbers, architects or designers.

The earliest maker or DIY date from the 1920s, the amateur hobbyists [62]. They relied on handbooks emphasizing imagination and an open mind technical aspects of radio communication [57]. They would often meet in person to discuss their work and ideas and other social matters. Later in the 1980s, a piece of low-cost MIDI equipment, which enabled people to record electronic music without any form of formal training, was developed, which resulted in the formation of the rave culture in the 1990s [62]. During this time, communities

were formed, where the hobbyists would create, explore, and exploit software systems resulting in the Hacker culture [62]. To this day, the "maker culture" attracts people across different skills and career levels to make something with their own hands, ranging from calligraphy, furniture to technology [64]. The maker culture emphasizes an active learning approach, where people learn new skills through getting their hands dirty in a social setting. The movement also emphasizes a networked, peer-led, informal and collaborative learning which is driven by fun and self-fulfilment. The culture encourages the new application of technologies, exploring the relationship between traditionally separated domains and ways of working, including film-making and computer programming [25].

2.6.1 Maker Communities

Within the Maker Movement, collaborative learning is realized though maker communities. These communities exist to allow people to meet and share ideas, skills and knowledge, and start making. Maker communities sometimes contain maker equipment such as 3D printers and prototyping equipment. Examples of these communities include Makerspaces and Makerfaires [65]. Networked technologies such as social media tools, websites have also played an enormous role the rise of the Maker Movement. These platforms have been used in the Maker Movement as a "show-and-tell", where makers share their projects, usually with pictures illustrating step-by-step building of the projects. These technologies have also formed the basis of knowledge and skillset sharing, and a central channel where ideas and information are easily shared and exchanged to large audiences [64].

2.6.2 Arduino and Raspberry Pi in the Maker Movement

The Arduino project, which is part of the open source movement as I had mentioned earlier has gained popularity in the Maker Movement over the years [54]. The most significant thing about Arduino is that it is easy to use, with great support available online. Alongside Arduino is the Raspberry Pi project. Raspberry Pi was released in 2012, and the main reason behind its release was to develop a basic, low-cost computer to teach programming concepts [92] [54]. An

interesting feature about the Raspberry Pi was that, similar to the Arduino, it also consists of General Purpose Input/Output (GPIO) which enables to work with sensors and actuators in Raspberry Pi projects [92]. As a result, the Raspberry Pi has gained recognition and success, especially in the Maker Movement. It has been used to create home automation, smart irrigation other fun projects projects [54].

2.7 Conclusion

This chapter covered a rich background to this research project. I discussed the emerging 4IR, its technological underpinnings, and its impact in the agricultural sector. I also explored the adoption of IoT in developing countries, the usability of its components, as well as some of the existing IoT smart irrigation systems. Finally, I introduced the open source and the Maker Movements, homes of the Arduino and Raspberry Pi projects as well as the very numerous software artefacts I will use throughout the rest of the work reported in this thesis. The next chapter focuses on an initial design of a smart irrigation system through a classical top-down approach, using a series of scenarios and directly based on the material discussed in this chapter.

Chapter 3

System Analysis and Components Selection

This chapter provides details on the initial top-down work done in this research. I designed the system through a series of scenarios relying on a top-down approach, which allowed us to identify what abstract components I needed to build the solution. I then mapped these components into off-the-shelf and well-supported hardware and software concrete implementations.

3.1 Functional Architecture

3.1.1 Nodes Functionality

Figure 3.1: An illustration of the nodes put in a field

Let us imagine a field that has a big enough extension to have varying properties, such as the soil type, sun exposure, as well as different terrain gradients. In this case, I want to monitor the soil moisture of every single different patch, according to its specific water requirements. This means that I have to put elements, or 'nodes' as illustrated by the arrows in Figure 3.1, to sense the soil moisture on every patch and automatically switch on and off water valves.

Based on functionality, each node should be equipped at a minimum with a soil moisture sensor, an electrovalve as an actuator and a programmable unit that periodically processes the input from the sensor, and decide if the water valve should be open or closed as illustrated in Figure 3.2.

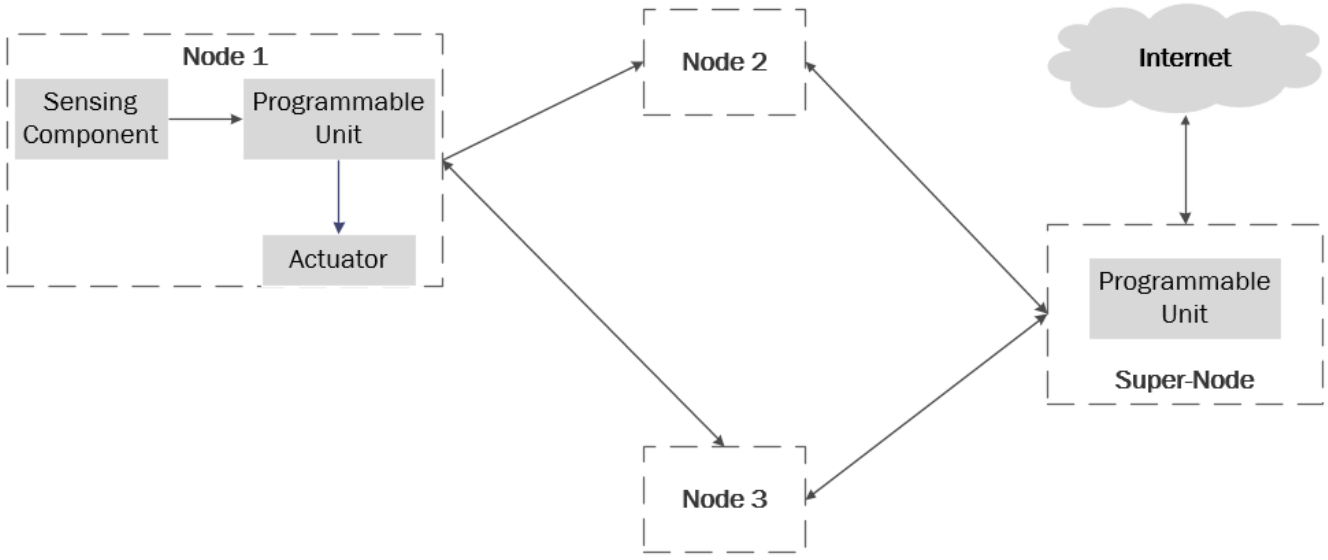

Figure 3.2: A small mesh network

Let us now imagine that two patches go to dry mode at the same time, and there is not enough water to irrigate both patches. I want the system to be able to water the patches based on priority, considering the type of plants in each patch and prioritize plants that are most sensitive to lack of water. In this case, instead of each node taking a decision in isolation, I want the nodes to be able to communicate to some intelligent "super-node" which orchestrates the irrigation activities. With this architecture, the system nodes measure soil moisture data, and relay the data to the super-node, which then decides on the best watering action, taking into account factors that go beyond the reach of a specific node.

3.1.2 System Intelligence

Natural weather conditions are of course a factor that I cannot avoid in building an irrigation system. So, let us now further imagine that a node goes to dry mode just before a heavy storm. Watering the plot would result in wasting water resources, and possibly even lead to soil erosion and plant flooding. Also, there might be local constraints such as a low dam that needs to be managed. I therefore want the system to be able to utilize connection to the bigger Internet to access weather forecasting (e.g precipitation) to provide an improved decision support for the

irrigation so that the system takes decisions not only based on the soil moisture and plants' water needs but also considers the weather forecasting in the area. In this manner, when the node reports dry state just before a rainfall, the system should postpone the irrigation by publishing an actuation to keep the node valve closed until the node reaches a dry state again. A very simple illustration of how the client manages a node irrigation based on both the sensor data and the local weather forecast is in Figure 3.3.

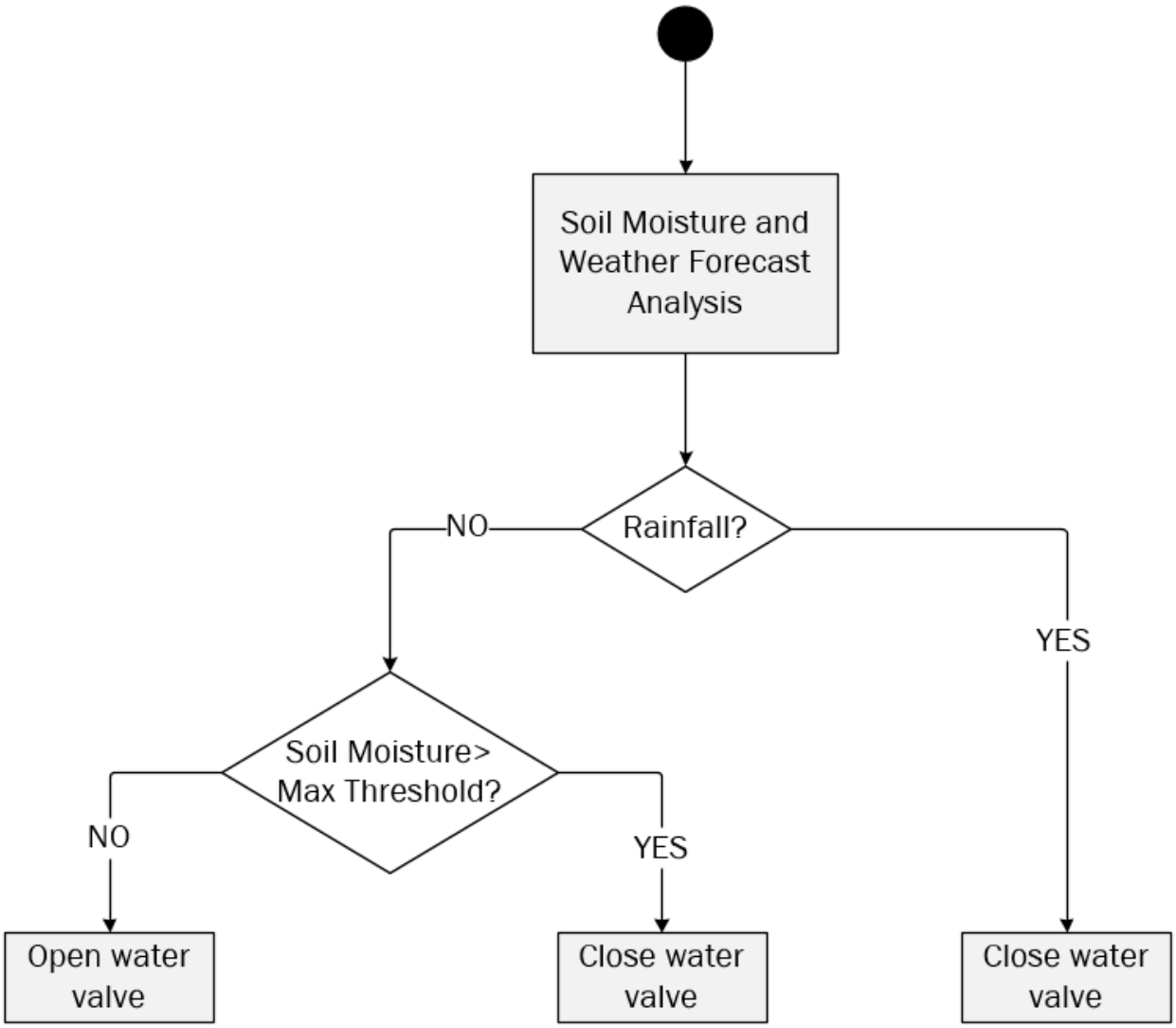

Figure 3.3: A simple illustration of the system intelligence

Adding this ability for our system to communicate with a bigger network realizes a complete IoT architecture, adding a cloud computing component. Cloud computing can apply data analytics such as visualization schemes to show device performance, identify inefficiencies, and come up with ways to improve our system. It also offers capabilities to draw patterns and correlations on historical data which can further contribute to creating algorithms to improve our system automation, through machine learning.

3.1.3 Communication between the Nodes

The communication between the system nodes necessitated by the first scenario above can be achieved in either a wired or a wireless network. Wired networks use physical wires to connect devices to the network, with each device connected by a separate wire and transmitting at the same speed. Wired networks are considered secure: tampering with the network would require physical access to the network, which would mean breaking physical boundaries. However, wired connections are expensive to install, considering the cost of the wires, installation and labour. Let us imagine that I wanted to add another node in our network, to irrigate a new patch. To extend the wired network (i.e. add the new node) would require additional wires, and labour costs. Similarly, if for example, two nodes in two different patches give the same soil moisture readings all the time, it makes no sense to keep the nodes in their position, as one becomes redundant, and so I would want to physically move one node to sense a different patch. Such a task would be difficult in a wired network: to change any device location would require additional cables to connect the device to the network.

Wireless networks, on the other hand, allow wireless enabled devices to communicate over radio signals, thus eliminating the need for physical wires. They require less equipment which makes them generally cheaper, easier and faster to install compared to wired networks. They do not restrict their connected devices to a fixed area, devices can access the network anywhere within range.

3.1.4 Mesh Communication

Connecting the nodes to a "super-node" in a wireless network poses the question of how to connect every single node to the super-node. One way of doing this would be to place the nodes such that they are all "visible" to the super-node (visible in this case means reachable radio-wise, considering the frequency and power of the radios used). However, this would mean that the wireless network size can only go up to a circle surrounding the super-node in the assumption that there are no obstacles within the circle. If I move a node outside the circle of the super-node, due, for example to the reasons I have stated earlier, then the node will no longer be able to communicate with the super-node, and will no longer be able send its soil moisture readings or receive actuation commands from the super-node. A wireless network solution supporting a "mesh network", where each one of the system nodes is able to communicate to the node next to it in order to reach the super node, would be useful. An illustration of this mesh networking idea can be seen in Figure 3.2, where node 1 wants to communicate with the super-node, and node 1 is not visible to the super-node. Data can then transmitted to the super-node via an intermediate node 2.

A solution that offers automatic route discovery capabilities would also be useful. Route discovery establishes a path made up of one or more intermediate nodes, where a message from a source node hops between the nodes until it reaches the destination node, where the transmission route is made up of one or more nodes acting as intermediate nodes. A solution that offers automatic route discovery would then mean that the solution itself takes care of establishing a path between a source and destination node through available intermediate nodes. What if an intermediate node "dies" (node 2 in our example)? Automatic route discovery would be useful, making possible to implement "self-healing": a new path through another available intermediate node (i.e node 3) will be established (if such a node exists naturally).

3.1.5 Data Management

I am aware that due to the way I designed the systems, I will have multiple data sources and consumers. For example, I have the nodes transmitting their soil moisture data to the super-node, which makes them data sources, and the super-node a data consumer. When the super-node transmits actuation commands to the nodes, the super-node becomes a data source, and the nodes become data consumers. When the super-node receives weather forecast information, then the super-node becomes the data consumer, and the cloud a data source. I am therefore looking for a solution that will manage the data flows i.e. ensure the right data is transmitted to the correct data consumer. A solution that decouples the system nodes from the super-node would be useful.

3.2 Abstract Component Selection

In this section I identify and discuss the general component categories that I would need to build the solution, based on the functional analysis in Section 1. An illustration of how these components are integrated is shown in Figure 3.4.

3.2.1 Soil Moisture Sensor

As mentioned earlier, it would be useful to equip each node with at least sensing component. A soil moisture sensor is a low-cost device which is used to detect soil moisture value [20][89]. The sensor can measure the water contents on the soil based on resistance or capacitance changes [58] [58] [77]. Resistive soil moisture sensors utilize the relationship between resistance and water content of the soil to gauge moisture levels [77]. Such a sensor consists of two probes that are directly inserted into the soil [77]. When there is more water, more current is conducted by the soil, hence resulting in lower resistance, and a report of a higher moisture level [6].

Capacitative sensors, on the other hand, work by measuring the change in capacitance, which is associated with a different dielectric permittivity relating to the water content of the soil [77] [45]. The sensor is built with a positive and negative plate, separated by a dielectric

medium in the middle [77]. Capacitative sensors are corrosion-free, allowing for better reading of moisture content, but they are typically more expensive than resistive ones [6]. Sensors generally include of an Analog-to-Digital Converter (ADC), which converts analog signals into a binary number, allowing them to transmit the soil moisture value as a number.

3.2.2 Actuator

The second component needed in each node is an actuator. An actuator is a device that produces physical movements based on energy and appropriate signals [69]. Linear actuators make a backwards or forward movement on a set linear plane. Rotatory actuators on the other hand revolve in a circular plane, and are not limited to a set path, they can rotate in the same direction for a long as they need [69]. Actuators are also classified based on their power supply source, electrical, pneumatic and hydraulic [31]. Pneumatic and hydraulic actuators consist of a piston inside a cylinder, and are mostly suitable for very large force linked with large motion, which is not the case in the system I am building [31]. Electrical actuators on the other hand use electric power to activate electrical devices such as valves and motors [31]. These actuators are generally used for on-off type control action, using electrical signals. This is the case of water electrovalves, typically used in automatic irrigation.

3.2.3 Programmable Units

Micro-controller

A node needs a programmable unit, in the form of a micro-controller. A micro-controller is a small, low-cost and self-contained computer on a chip. Micro-controllers are often embedded inside consumer devices to control the device functions [76]. Micro-controllers cost less to produce than micro-computers, have a lower power footprint (so they might operate on battery if necessary), and can be used with a vast range of electronic equipment through their I/O features; a sensor and actuator in our case [76] [52]. Unlike micro-computers, micro-controllers operate without an operating system, which gives full system control to the programmer and less chances of breaking.

Micro-controllers are classified into categories according to their arithmetic logic size (in bits), memory, architecture and instruction set characteristics [11]. The micro-controller arithmetic size indicates the maximum binary data size that the micro-controller can handle as a single unit [52]. These differences are highlighted during mathematical operations. A higher number of bits improves the micro-controller performance and accuracy [52]. However our system does not require high computation capability, and therefore, even a micro-controller with minimal arithmetic logic size (8 bits) will be sufficient for us.

Micro controllers support machine instruction sets using Complex Instruction Set Computer (CISC) or Reduced Instruction Set Computer (RISC) [11]. The CISC architecture aims to complete a task with minimal lines of assembly and allows programmers to use one instruction to execute multiple low-level operations (e.g. load from memory, or a memory store) [84]. With the CISC architecture, the processor speed is compromised because a single instruction may take more than one clock size to execute, and also, the architecture increases the program complexity [42]. RISC, on the other hand, uses simple commands that can be broken down into several instructions that achieve low-level operations in one clock cycle [84]. The CISC architecture increases the processor speed, however, they have large memory caches on the chip itself [42]. Again, because the system I am building does not require much computation, these architectural differences are not significant to us.

Micro-controller memory architecture comes in two forms, Harvard and Princeton [11]. Harvard memory architecture physically separates instruction and data, paths from the memory pathway, which means that the CPU can read instructions and perform data memory access, both at the same time [86]. This architecture improves the program execution speed at the cost of more hardware complexity [86]. Princeton memory, on the other hand, stores both instruction and data in the same memory, and so uses the same path to fetch instruction and data [86]. This means that the micro-controller cannot simultaneously read an instruction and perform data memory access, which then makes the Harvard memory architecture faster

than the Princeton architecture [86]. These architectural differences are important in specific settings, but not in the system I am building.

Micro-computer

Although using a micro-controller is a good choice for the system nodes, micro-controllers operate on limited power and memory, and can typically only run a single program at a time. On the other hand, the super-node requires a programmable unit that is more powerful than a micro-controller, with more processing power, and an operating system. I therefore decided to use a micro-computer as the programmable unit in the super-node. A micro-computer is a small, relatively inexpensive computer. Similar to a typical computer, a micro-computer consists of a Central Processing Unit (CPU), memory unit and Input/Output (I/O) unit [37]. A micro-computer CPU is called a microprocessor, and performs all arithmetic and logic operations. A micro-computer memory unit is implemented using some form of Read-Only Memory (ROM) and Random Access Memory (RAM) chips [37]. RAM memory stores the CPU data and programs, and it is volatile. ROM is non-volatile memory that permanently stores computer instructions supplied by its manufacturer [37]. The I/O unit provides interface for input and output devices such as keyboard, mouse and HDMI screen [37]. Unlike micro-controllers, micro-computers can run a fully-fledged operating system [37].

3.2.4 ZigBee Protocol

As stated in section 1.2, I want the system nodes to be able to communicate with a super-node as easily and reliably as possible. I was looking for a wireless communication solution that offers mesh capabilities, as well as automatic route discovery and advanced self-healing capabilities. A good candidate for such a solution is the ZigBee protocol or mesh WiFi. However, mesh WiFi routers are expensive compared to ZigBee devices. In addition, WiFi devices generally consume more power than ZigBee devices, which makes them less suitable for constrained environments such as the one I am working with. These characteristics eliminate mesh WiFi as an option for the system I am building. This leaves us with the ZigBee protocol. ZigBee is a

short-range wireless communication protocol based on the IEEE 802.15.4 standard developed to allow secure, low-cost, low-power wireless machine-to-machine(M2M) communication and Internet of Things (IoT) networks [30]. ZigBee operates at a frequency of 2.4GHz, with a 250kbps data rate, and covers a 10-100m range [63] [17] [78]. An important ZigBee feature is the ability to support mesh networking, additional to the start, and tree network topologies supported by the IEEE 802.15.4 standard [78] [63] [30]. ZigBee offers automatic route discovery capabilities, where the route is made up of one or more intermediate nodes as I had discussed in Section 1 [78]. ZigBee also offers advanced "self-healing" capabilities, which further solves the problem of nodes communicating with the super-node even if an intermediate node fails or detaches from the super-node, by establishing a new route through an available intermediate node where possible [63]. The protocol provides low-power sleep mode, where the devices in this technology can switch to low power sleep mode when they are not involved in data exchange [78]. This capability is useful to implement the idea of devices going to sleep between transmitting soil moisture data to, and receiving actuation commands from the super-node as I have stated earlier.

3.2.5 MQTT Protocol

MQTT is a data transfer protocol typically running on top of the TCP/IP stack, and originating from early IoT efforts [41]. MQTT was designed as a lightweight messaging protocol to reduce network bandwidth for constrained environments, such as low power, limited computation capability and memory, and limited bandwidth [18]. MQTT implements the publish / subscribe pattern, which is based on a message broker/server and all other nodes placed around the broker in a star topology. The publish / subscribe pattern means that the subscribers act independently from the publishers and vice versa, realizing the decoupling between the data sources and consumers I discussed at the end of section 1 [96]. The publishers can publish data to the server even if the subscribers are not available, for example sleeping [63]. Similarly, the subscribers can retrieve published from the broker even if the publishers are not available. [63]. This establishes an asynchronous communication between the publishers and subscribers [96]. The communication of the subscribers and publishers with the server is, on the other hand,

synchronous [96]. Another important feature of the publish/subscriber messaging pattern is that one publisher can send data to different subscribers [96]. Because the data is published to the server, published data can be accessed by multiple subscribers without the need for the publisher to send the data to every subscriber explicitly [63]. Also, one subscriber can access data from multiple publishers [96].

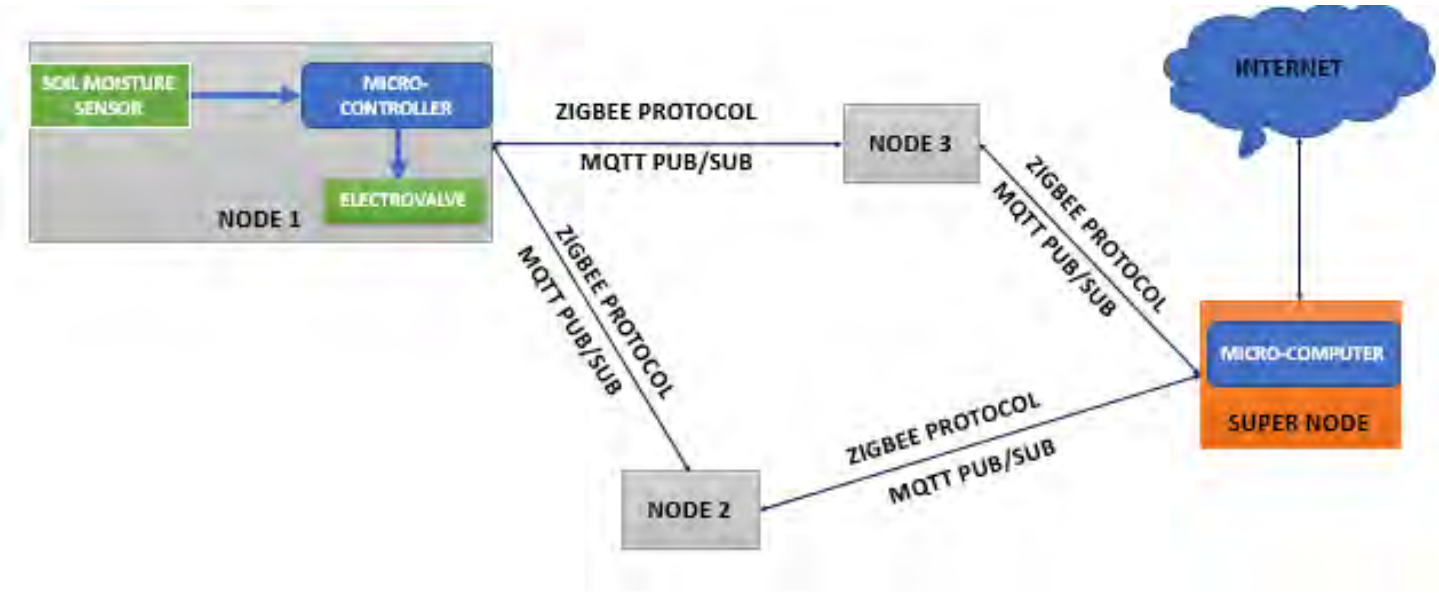

Figure 3.4: An illustration of the system abstract components

3.3 Concrete Components Selection

In this section I identify the specific off-the-shelf and well-supported hardware and software entities that I will use to build the solution, for each of the abstract categories described in section 2.

3.3.1 MH-Series Soil Moisture Sensor

I decided for the commercially available MH-Series soil moisture sensor as illustrated in Figure 3.5. The sensor is a resistive soil moisture sensor, i.e., it utilizes the relationship between

electrical resistance and water content to gauge moisture levels of the soil moisture. The soil moisture module consists of two parts, the main sensor and the the electronic part which includes an analogue to digital converter (ADC). The sensor consists of two probes which interact with the soil to measure the volumetric content of water in the soil. The two probes inject the current in the soil and identify the soil resistance, which is then converted to a moisture value [40]. The sensor's ADC uses 10 bits, and so, values are mapped from 0 to 1023 [40]. The sensor is equipped with both digital and analog output, making it able to work both in digital and analog mode [20].

3.3.2 Electrovalve

I decided to use an electrovalve as an actuating component to control watering patches i.e switch them ON/OFF the water valve based on the soil moisture sensor reading. An electrovalve can either partially or completely blocks a pipe to change the amount of water that flows through it. For now, I am utilizing the idea of completely switching ON/OFF the valve. For implementation purposes, however, I used a simulation of the electrovalve using two LEDs, where one LED lights up when the soil is dry, and the other one lights up when the soil is wet.

3.3.3 Arduino Uno

I decided to work with the Arduino family of micro-controllers. Arduino is an open-source hardware development platform [19]. The platform is said to provide a simple and accessible user experience through its hardware and software components [102]. For example, the Arduino software is made simple and easy for beginners, at the same time it is flexible enough for advanced users [102]. The Arduino platform is widely used in the Maker Movement and in the open-source movement (as described in Chapter 2), which means that there should be great support and the tools should be mature enough for use as described in Chapter 1. As I have stated, the nature of the system I am building does not need extensive computational capabilities, or even much memory, since I anticipated to have fairly small Arduino programs, and so I just need a micro-controller that will have just enough tools. Within the Arduino family

of micro-controllers, I selected the Arduino Uno. The Arduino Uno is highly recommended by the Arduino community, it is the most documented of the whole Arduino family, and so I assumed to have great support and help to build our system with this micro-controller [9]. The Arduino Uno consists of an 8 bit RISC processor, 32 kB flash memory, 0.5 kB is used for the bootloader [19]. It also include 14 I/O digital pins (out of which 6 can be used as Pulse Width Modulation (PWM) outputs), which can be used to connect the soil moisture sensor and the electrovalve to the micro-controller [19]. The board also includes 1 UART serial port, which can be used for serial communication with the XBee device (see Figuree 3.5) [19]. The board can be powered with a 9V battery, which makes it portable in various experimental settings and finally in a deployment.

3.3.4 Raspberry Pi

I decided to use a Raspberry Pi as a micro-computer for our system. Raspberry Pi is a family of low-cost, open-source hardware platform or micro-computers [91]. Raspberry Pi micro-computer can operate with various operating systems, including Raspbian - a free open-source Linux distribution, which then became useful for maintaining an affordable platform [49]. Like the Arduino micro-controller, the Raspberry Pi family has made its mark in the Maker Movement, which means that there is good support available for the solution of the type of problem at hand [91]. Within the Raspberry Pi family, I decided to work with Raspberry Pi 3 (at the time I initiated the project, Raspberry Pi 4 was not yet released). Raspberry Pi 3 is based on a 64-bit quad-core CPU, with a clock speed of 1.2 GHz [12]. It includes 1GB RAM, 4 USB ports usable with peripherals such as keyboard, mouse, and an HDMI port to use the computer with a screen [12]. Raspberry Pi 3 also includes two UART serial ports usable for serial communication with the XBee device as I have shown in Figure 3.5 [12]. It also includes an Ethernet port, through which I can easily connect the computer to the Internet, which is an essential requirement for the super-node as detailed earlier [12].

3.3.5 ZigBee XBee

I decided to use the commercially widely available family of XBee RF modules produced by the Digi International Company. These modules are indeed commonly used to provide connectivity to electronic devices such as micro-controllers and sensors [24]. Digi XBee modules can be easily connected to an intelligent device, the Arduino UNO micro-controller and Raspberry Pi in our case, through the UART serial interface [24]. Alternatively, they operate as stand-alone through their programmable variants, which eliminates the need for a micro-controller [24]. These modules require little or no additional development and are easily configured [24]. These modules have also made a mark in the large and strong Maker and open-source community, which of course means that there is great support to help get started, and, again, I wanted to stay within these movements [24]. Digi XBee RF modules are categorized from Series 1 up to 3, with varying capabilities [28]. XBee Series 1 RF modules, also known as XBee 802.15.4 implements the IEEE 802.15.4 stack, and supports point-to-point communication and proprietary mesh networking [28]. These modules are great for simple, small sized systems, they act as a simple cable replacement [28]. The XBee Series 2 modules or XBee ZigBee modules are built for more complex and larger networks, they implement the full ZigBee stack which is built on top of the IEEE 802.15.4 stack, and support ZigBee mesh networking, with improved range and less power consumption compared to Series 1 modules [28]. XBee Series 3 modules add new capabilities such as Digi TrustFence security framework, Bluetooth low energy (BLE) local commissioning and introduces MicroPython, which eliminates the need for an external micro-controller [24]. I decided to work with the Digi XBee Series 2 RF modules, since they are sufficient to implement the ZigBee mesh networking, route discovery, and self-healing capabilities [28].

3.3.6 MQTT Messaging Protocol

I looked for an MQTT implementation to implement the MQTT publisher/subscriber model. Preferably, the platform should be able to support the Raspberry Pi and Arduino UNO micro-controllers. A good candidate for this was the Eclipse Mosquitto MQTT broker [26]. Mosquitto

implements an MQTT broker and a C library of MQTT clients, carrying out messaging using a publish/subscribe model [26]. Mosquitto is open-source, which means that there must be great support available, and the tools must be fairly matured for use [26]. Mosquitto is lightweight and is able to work for a wide range of devices, from low power single boards to full computer servers including the Raspberry Pi [26].

As I had stated in Section 2, that MQTT requires sophisticated transport layer protocols such as TCP/IP, not provided by ZigBee protocol I planned to use. A good candidate appeared to be a variant of MQTT, known as MQTT-SN. MQTT-SN was specifically designed to operate on low-power and low-cost devices, to add the possibility of using sensor networks with minimal power resources and constrained processing capabilities such ZigBee, without compromising the MQTT subscriber/publisher messaging pattern [80]. The protocol introduces a gateway between MQTT-SN clients and the MQTT broker [80]. The gateway performs protocol conversion between MQTT-SN clients running over a non-TCP/IP stack, and the MQTT broker running on top of the TCP/IP stack [80]. I therefore decided to explore this MQTT-SN protocol, specifically to identify how mature tools was and how easy it would be to build our solution with it. As I will see in Chapter 6, this choice had to be reversed and I ended up building our own gateway, yoGa

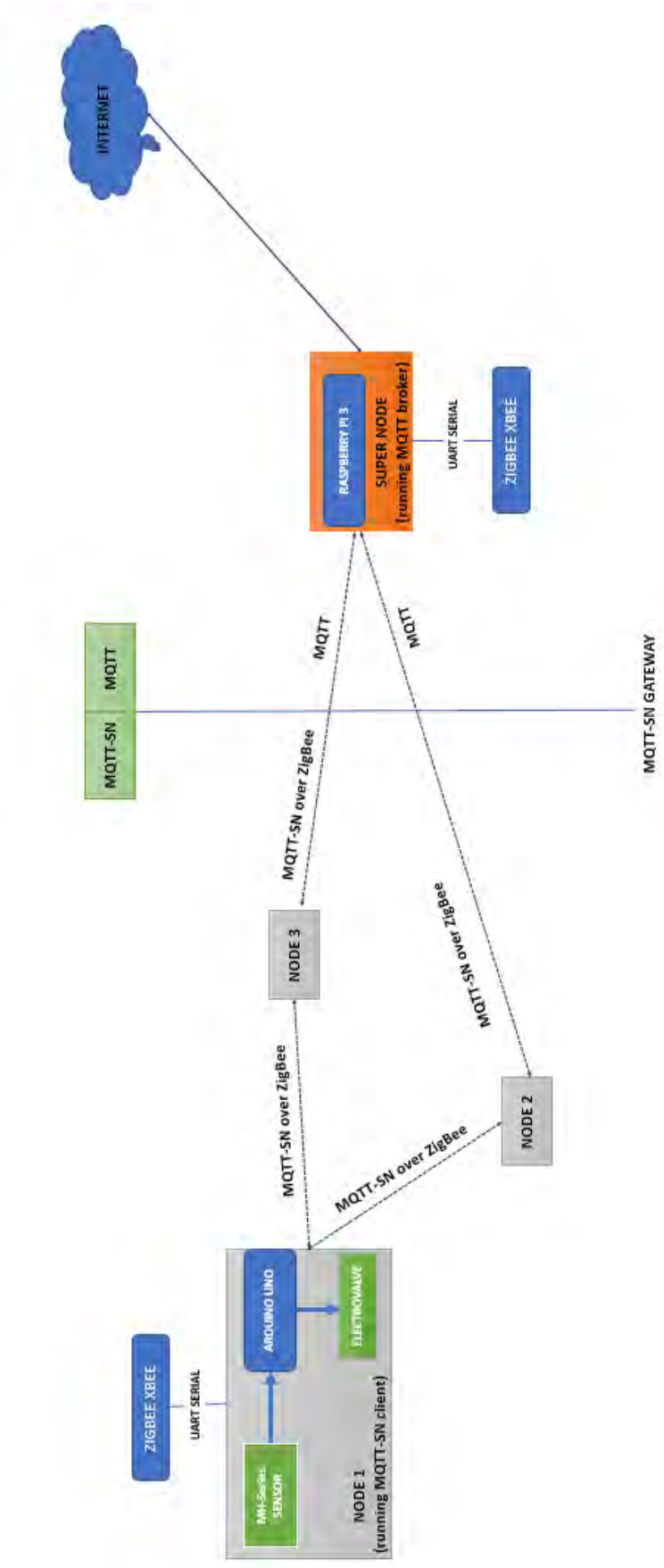

Figure 3.5: An illustration of the full system components

3.3.7 Fail Safe Considerations

In section 1 of this chapter, I explored the solution I wanted to build through a series of scenarios. Intentionally, these scenarios did not consider failures, although not core to the aim of this thesis. This section discusses possible failures that might take place and presents simple ways to ensure the system is still able to perform its basic non-smart irrigation activities. The core idea behind the realization of simple remedies to failure is to add multiple layers of decentralized capabilities.

Let us imagine that a node detaches from the super-node. Of course the self-healing capabilities I am implementing would ensure that the node is still able to talk to the super-node through an available intermediate node as I have discussed earlier. But what if the intermediate node dies, and there is no available intermediate node? Obviously, this would mean that the node is no longer able to relay its soil moisture sensor readings to or receive actuation commands from the super-node. To cater for such a scenario, if a node is unable to reach the super-node for maybe two transmissions, then the node should be able to act independently and handle its own irrigation activities, i.e. switch ON/OFF the water valve based on its soil moisture readings. Similarly, if the super-node itself dies, then all the nodes should be able to act independently and continue irrigation. However, in addition to the response, I want the nodes to also send alarm to a human being for intervention, which may require additional hardware in general. Similarly, if the super-node loses connection to the Internet, the node should then make irrigation decisions based on the soil moisture data it receives from the nodes. Although this approach prevents the system from going chaos, it eliminates the intelligent part of the solution, that is, incorporating the weather information and the precision algorithms.

Finally, there could be situations where nodes are not performing properly, for example, the node always reports the patch as being "dry" even if the super-node transmits an "open valve" actuation commands. In this case, the sensor could be broken, and hence giving incorrect readings, or the electrovalve could be broken and thus unable to open even when instructed to do so. In this case, I want the system to send an alarm to a human being for intervention.

3.4 Conclusion

This chapter reported on the initial top-down work done in this research design the solution through a series of experiments. In this top-down work, the abstract components required to build a smart irrigation system were identified. Following this, the off-the-shelf and well-supported hardware and software tools that implement the identified abstract components of the system were selected. The next chapter reports on the implementation of a smart irrigation system with the selected hardware and software tools.

Chapter 4

Experimenting with Arduino and XBee

This chapter starts the implementation of the smart irrigation system using the hardware and software tools selected in the previous chapter. The implementation follows a bottom-up approach through a series of experiments. I experimented with each tool in isolation, and then integrated the tools. This approach deepened my understanding of each tool, while building a growing prototype of the smart irrigation solution. This chapter first reports on a simple set of tutorials, some of which are taken from the Arduino Starter Kit booklet, then moves to configuring and testing XBee devices using the XCTU program, first in a P2P network, a star network and then a ZigBee mesh network.

4.1 Experimenting with Arduino

4.1.1 Aim

The aim of this experiment was to understand how to use the Arduino software to interact and control the Arduino hardware components. In this experiment I made use of the tutorials provided by the Arduino Starter Kit booklet [74]. In each Tutorial documented in this section, I followed close instruction booklet to build the electric circuit, as well as programming the Arduino board.

4.1.2 Tools

- Arduino Uno R3 board
- Arduino Integrated Environment (IDE)
- Light LEDs, button and resistors

4.1.3 Getting Started with Arduino

The Arduino tools involve both hardware and software components. The software, called the Arduino Integrated Environment (IDE), resides in a desktop, and is used to write Arduino program called sketches, and upload them to Arduino compatible boards. These programs are used to interact with and control the behaviour of the hardware components. As a result, before I could start experimenting with the Arduino, I needed to first download and install the Arduino IDE. The IDE is an open-source application running on various platforms such as Windows, Linux and Mac OS and available from: arduino.cc/download. I then downloaded and installed the latest Windows installer of the application on the computer. The Arduino is connected via a USB. Another important step before getting started with experimenting the Arduino hardware and software components was to ensure that the Arduino IDE was configured for my specific Arduino board, and that the serial port that I would be using was set properly. To do this, I navigated to the "Tools" tab in the software. From the tools tab, I selected the "Board" option on the drop-down menu and selected my specific Arduino board "Arduino/Genuino Uno". To set the serial port, I navigated again to the "Tools" tab, selected the "Port" option on the drop-down menu, and selected the "COM4" serial port. The Arduino IDE provides great ease of use in uploading the Arduino sketches to the board, such that one only needs to configure and specify the type of Arduino board they are working with, the board serial port, and then in one button click, the code is uploaded to the Arduino board [83].

4.1.4 Arduino Programming

As previously stated, Arduino programs are called sketches, and every Arduino sketch typically consists of two main parts, the *setup* and *loop* functions. Both of the functions are of type void. The *setup* runs once, when the board is switched on, while the *loop* code runs after the setup code, and is repeated up until the board is switched off. As previously stated, Arduino allows code uploading in one button click, the *upload* button: the code is compiled and uploaded into the Arduino board. Naturally, during the compilation and uploading of the code, the software runs various events and processes in the background hidden from the user, to maintain the ease of use. For example, when the button is clicked, the code is sent to the Arduino AVR-GCC compiler [97]. During the compilation, the object code is linked against the standard Arduino libraries pre-compiled header files. The resulting .hex file is then transmitted into the circuit flash memory using the USB cable [83].

Arduino Electric Circuit

Arduino use a "breadboard" to prototype and build the projects electric circuits. A breadboard typically consists of holes with metal clips underneath and organised in "half-rows" (5 holes) which are electrically connected. These half-rows are the "playground" for building the circuit in that it is where the hardware components are inserted. On each side of the row are additional buses (voltage and current bus) which are used to supply electric power to the circuit ("half-rows") when connected into a power supply.

4.1.5 Tutorial 1

This tutorial was a small example that illustrated the use of the Arduino digital pins to control Arduino LED lights and button. The program uses three LED lights (one green and two red) and a button. When the board is powered, the green LEDs lit. When the button is pressed, one red button is lit, and then, after a quarter a second, the other red button is lit.

Additional Components

- Green LED
- 2 Red LEDs
- Button
- 220 OHM and 10 KOHM resistors
- Jumper wires

Building the Circuit Prototype

- I then placed the green and two red LEDs on the breadboard. LED lights contain two legs, a longer leg called the anode, and a shorter leg called the cathode. I attached the green and two red LEDs cathode to the 220-ohm resistor and the anode to the Arduino digital pin 3, 4 and 5 respectively.
- I placed the button on the breadboard, and connected the one side to power, and the other side to the Arduino digital pin 2.
- The Arduino Breadboard was powered by connecting the breadboard ground bus to the Arduino ground pin and the power bus to the Arduino 5V using jumper wires.

Programming the Arduino

As I had mentioned earlier that Arduino uses the software to control the hardware components. After building the circuit, I proceeded to writing the Arduino sketch to control the behaviour of the components. As illustrated in Listing 4.1, in the setup function, I initialized the digital pins in which the LEDs and the button are connected, using the *pinMode* function. The function takes two arguments, the pin number and the pin mode (INPUT or OUTPUT). The LED pins were configured as output digital pins, and the button pin as an input digital pin. I used a global variable *buttonState* to hold the state of the button (pressed or not pressed). In the loop function, I used the *digitalRead* function to read the button state, and stored this value

to the global variable *buttonState* which I created earlier. The *digitalRead* function measures the voltage on a specified Arduino pin and returns a binary value - returns HIGH when the voltage read is above 3V, and LOW if it is less than 3V. And so, if the button is pushed, then the function returns HIGH. In contrast, if the button is not pushed, the function returns LOW. I then used the *digitalWrite* to switch on and off the LED lights. The function writes a binary value (HIGH or LOW) to a specific Arduino pin. Voltage readings to the button digital pin, so as to switch on and off the LED lights. The function writes a value (HIGH or LOW) to the Arduino pin.

4.1.6 Tutorial 2

This tutorial introduced the use of an analogue sensor to measure temperature using the Arduino analogue pins. This is achieved by making use of the Arduino built-in Analogue-to-Digital Converter (ADC). The program measures the current temperature and switches on LEDs based on the temperature readings. One LED if the temperature value is 22 °C or 23 °C, two if it is 24 °C or 25 °C, and I lit up all LEDs if the temperature value is 26 °C or more.

Additional Components

- 3 Red LEDs
- TMP36 Temperature sensor
- 3 220 OHM Resistors
- Jumper wires

As illustrated by Figure 4.1, I placed the thee red LED lights to the breadboard with each of the LED cathodes connected to ground using a 220-ohm resistor, and the anodes to pin 2,3 and 4. I then placed the temperature sensor on the breadboard. The sensor has two sides, a flat side and a more rounded side. I placed the sensor with the rounded side facing

```
int switchState = 0; //holds the switch state
void setup(){
  //configuring the digital pins
 pinMode(3,OUTPUT); //green LED
 pinMode(4,OUTPUT); // red LED
 pinMode(5,OUTPUT); //red LED
 pinMode(2,INPUT); //Switch
}
void loop(){
 switchState = digitalRead(2); // Reads the switch state
  if (switchState == LOW) {// the buton is not pressed

 digitalWrite(3, HIGH); // green LED switched on
 digitalWrite(4, LOW); // red LED off
 digitalWrite(5, LOW); // red LED off
 }
 else { // the buton is pressed
 digitalWrite(3, LOW);
 digitalWrite(4, LOW);
 digitalWrite(5, HIGH);
 delay(250); // wait for a quarter second
 // toggle the LEDs
 digitalWrite(4, HIGH);
 digitalWrite(5, LOW);
 delay(250); // wait for a quarter second
 }
} // go back to the beginning of the loop
```

Listing 4.1: Tutorial 1 Arduino sketch (from [74])

away from the Arduino, and then connected the right pin on the flat facing side to ground, the middle to the Arduino analogue pin 0 (A0), and the left to power using jumper wires.

Tutorial 2 circuit prototype

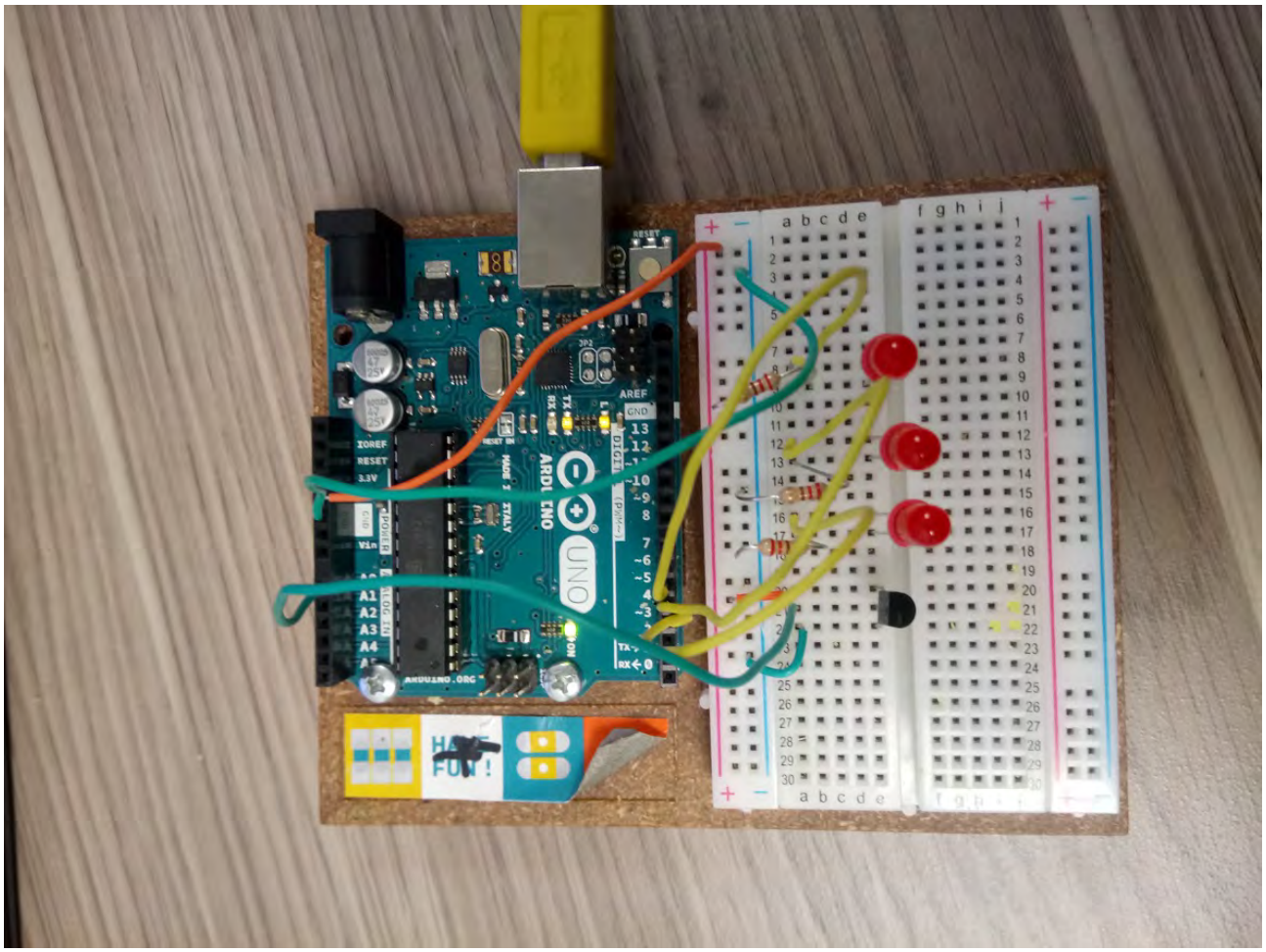

Figure 4.1: Tutorial 2 circuit board prototype

Programming the Arduino

As shown in Listing 4.2, I created two constant variables, *sensorPin* and *baselineTemp* to hold the number of the analogue pin in which the temperature sensor is connected and the baseline temperature, respectively. In the *setup* function, I opened the serial monitor at 9600 baud rate (speed at which are sent in a second) using the *begin* method. The serial monitor is a tool that

```
const int sensorPin=A0;
const float baselineTemp =20.0;

void setup() {
Serial.begin(9600);
//opens connection between Arduino and computer, so you can see
   analog input on computer screen
//9600 is the speed at which the two will communicate
//will use IDE serial number to view information from Arduino

for (int pinNumber =2; pinNumber<5; pinNumber++){
  //initializes all pins to be output pins, and turns them off
  pinMode(pinNumber,OUTPUT);
  digitalWrite(pinNumber,LOW);
 }
```

Listing 4.2: Tutorial 2 Arduino sketch: Variables declarations and *setup function* (from [74])

allows communication between the Arduino and the computer. I then initialised the LED pins as output using the *pinMode* function. that I used in the previous tutorial.

Inside the for loop, I read the sensor value from the analogue pin using the *analogueRead* function, and then sent off this raw value to the computer using the *serial.print* function as illustrated in Listing 4.3. I then performed basic math to convert sensor reading to a voltage value (V), which was also converted to an actual temperature value (°C). Both the voltage value and the temperature value were sent to the computer.

I then used an *if statement* to control the LED lights based on the temperature value using the *digitalWrite* function as illustrated in Listing 4.4. In the *if statement* I kept all the LED lights off if the temperature value is less than the baseline temperature constant variable (20,0 °C) that I created earlier. I lit up only one LED if the temperature value is 22 °C or

```
void loop() {
  int sensorVal =analogRead(sensorPin);
  Serial.print("Sensor Value: ");
  Serial.print(sensorVal); //Reads the sensor value and sends it off
    to the computer



  //converts the sensor reading into voltage, and displays it on the
  float voltage = (sensorVal/1024.0)*5.0;
  Serial.print(",Volts: ");
  Serial.print(voltage);
  Serial.print(" , degrees C:");

 //converts the voltage into temperature and prints it on a new line
  float temperature =(voltage -.5)*100;
  Serial.println(temperature);
```

Listing 4.3: Tutorial 2 Arduino sketch *loop* function: Reading the sensor values (from [74])

23 °C, two if it is 24 °C or 25 °C, and I lit up all LEDs if the temperature value is 26 °C or more.

4.1.7 Tutorial 3

This tutorial explored the use of soil moisture sensors in Arduino. Arduino program program measures the moisture reading through the analogue pins and displayed the calculated average value. The average is then displayed on the Arduino serial monitor.

Additional Components

- MH-Sensor-Series soil moisture sensor
- Jumper wires

Building the Circuit Prototype

The soil moisture module consists of two parts, the main sensor and the MH Sensor Series. The sensor consists of two probes that measure the volumetric content of water in the soil. The two probes inject the current in the soil and identify the soil resistance, which is then converted to a moisture value [40]. When there is more water, more current is conducted by the soil, and a higher moisture level is reported. In contrast, when the soil is drier, less current is conducted, and lower moisture value is reported. The MH Sensor part consists of pins, the VCC pin for power, A0 for Analogue output, D0 for Digital output and GND for ground connection. In addition, the sensor acts as an analogue to digital converting (ADC) of the sensor value. The sensor's ACD uses 10 bits, and so, values are mapped from 0 to 1023 [40]. I placed the sensor pins to the Arduino as follows (see Figure 4.2):

- VCC : Arduino 3V pin
- GND : Arduino ground pin
- A0 : Arduino A0 analogue pin

```
  if (temperature <baselineTemp){
    digitalWrite(2,LOW);
    digitalWrite(3,LOW);
    digitalWrite(4,LOW);
  }

  else if (temperature >=baselineTemp+2 && temperature <
   baselineTemp +4){
    digitalWrite(2,HIGH);
    digitalWrite(2,LOW);
    digitalWrite(2,LOW);
  }
 else if (temperature >=baselineTemp+4 && temperature < baselineTemp
    +6){
    digitalWrite(2,HIGH);
    digitalWrite(2,HIGH);
    digitalWrite(2,LOW);
  }
   else if (temperature >=baselineTemp+6){
    digitalWrite(2,HIGH);
    digitalWrite(2,HIGH);
    digitalWrite(2,HIGH);
  }
  delay(1);
}
```

Listing 4.4: Tutorial 2 Arduino sketch loop function: Controlling the LEDs based on the sensor value (from [74])

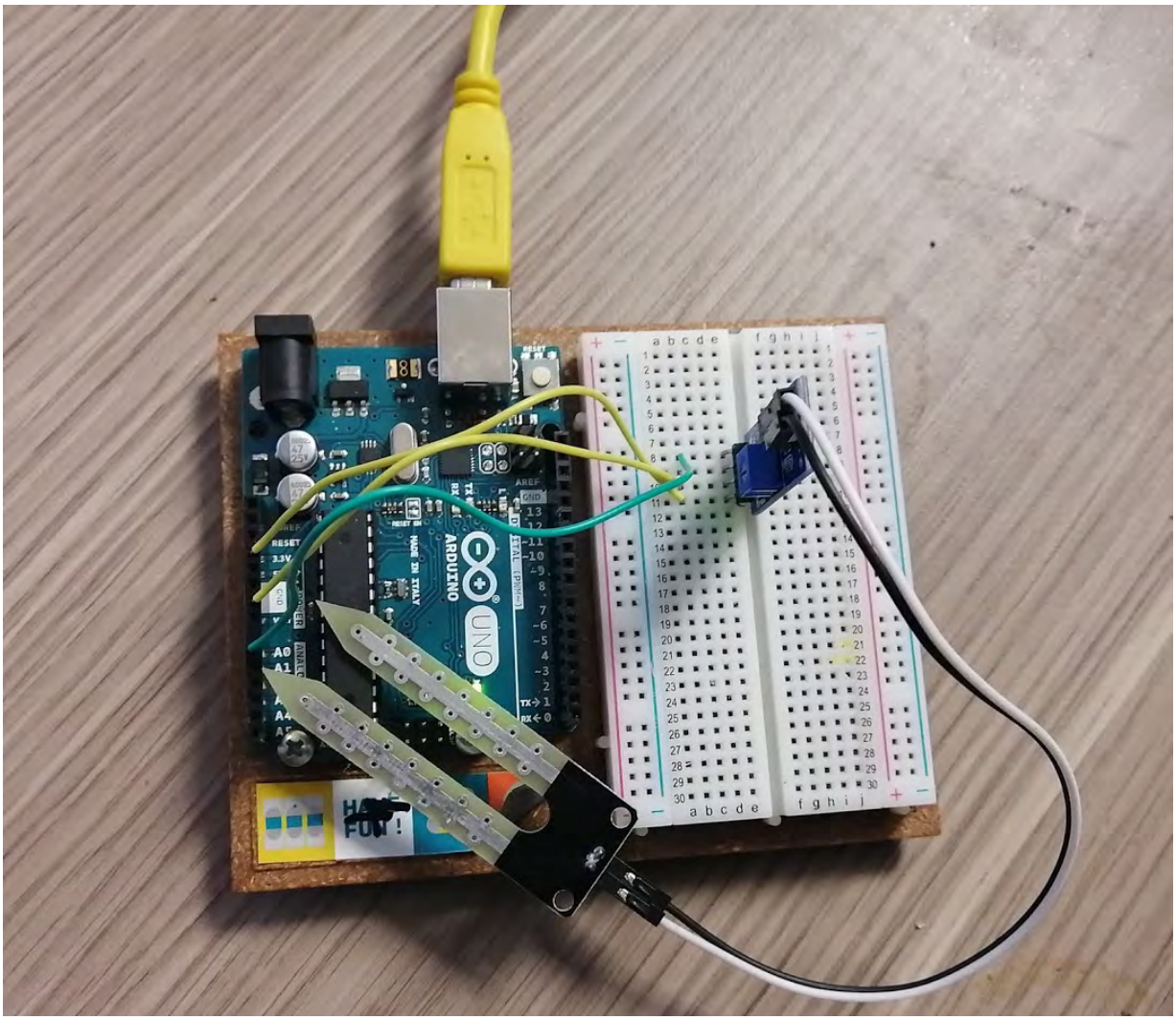

Figure 4.2: Tutorial 3 electric circuit prototype

Programming the Arduino Board

As shown in Listing 4.5, I created two global variables, one to hold the analog pin in which the sensor is connected, and the second the sensor reading. In the setup function, I simply initialized a serial connection and used it to display the sensor values later. The loop function is where I read and displayed the sensor values. I used the *analogueRead* function to read the sensor reading. The function converts the voltage range to a 10 bit value (0 to 1023).

```
int sensorPin = A0;
int SensorReading=0;
int limit = 300;
uint16_t SensorValue[1]={0};

void setup() {
 Serial.begin(9600);
 pinMode(13, OUTPUT);
}

void loop() {
   SensorReading = analogRead(sensorPin);
   SensorValue[1] = map(SensorReading,240,1023,0,100);
   Serial.println("Analog Value : ");
   Serial.println(SensorValue[1]);
   delay(3000);
}
```

Listing 4.5: Tutorial 3 Arduino sketch (adapted from [8])

4.2 Interfacing XBee with Arduino

4.2.1 Aim

The aim of this experiment was to understand how to configure XBee devices, and then make the Arduino boards to talk to each other using these devices.

4.2.2 Tools

- 3 x Arduino Uno R3boards
- 3 x Arduino XBee shield
- 3x XBee S2C modules
- 3 x XBee S1 modules
- Power Supply 3 x (USB A-B cable)
- Windows PC
- XCTU

I used the commercially available Series 1 XBee radio modules. As illustrated in Fig 6, the modules were connected to the Arduino boards through the Arduino XBee shield, a module that acts as an interface between the Arduino and the XBee radio modules. The modules were configured using the XCTU, an application for developers to control XBee radio modules via a well designed graphical interface. XCTU is free and can run on the main operating system in use.

4.2.3 Interfacing the XBee Modules

As I said earlier, the Arduino XBee shield provides an XBee interface in Arduino boards so that the boards are able to communicate wirelessly [13]. The shield is designed to work with any

development board consisting of the Arduino standard footprint. In addition, the shield works with series 1 and 2 standard and pro versions of the XBee modules. The shield provides an XBee socket, serial communication switch, status LEDs which indicate power and data reception and transmission. In addition, the shield consists of the Arduino board analogue, digital, ground and power pins that are hidden by the shield.

The XBee socket is where the XBee module is inserted so that the XBee module draws power from the Arduino 5V which is then reduced to 3.3V by the shield before being transmitted to the XBee. In addition, the XBee module Rx, TX and ground pins are connected to the Arduino board Rx, Tx and ground pin respectively. In addition, the shield serial communication switch determines the connection between the XBee serial communication and the micro-controller and FTDI USB-to-serial chip. The switch has two positions, the USB and XBee position. In the USB position, the XBee DOUT and DIN pins are connected to the FTDI chip Rx and Tx pins respectively. With this connection, the XBee module communicates directly with the PC. Alternatively, in the XBee position, the XBee DOUT and DIN pins are connected to the micro-controller Rx and Tx pins, respectively, while the micro-controller Rx and Tx pins are still connected to the FTDI chip Tx and Rx pins respectively. Figure 4.3 illustrates how the XBee module is fitted into the Arduino via the XBee shield. Interfacing the XBee modules with Arduino uses the Arduino board as USB-UART converter. To do this, I loaded an empty sketch on the Arduino board. This ensures that the Arduino bootloader is bypassed and that the board is only used for serial communication. (An alternative way to interfacing the XBee modules with Arduino is using an USB Xbee Adapter. The adapter would easily connect the XBee to the computer USB port. However, I picked Arduino mainly because I had Arduino boards at hand already.

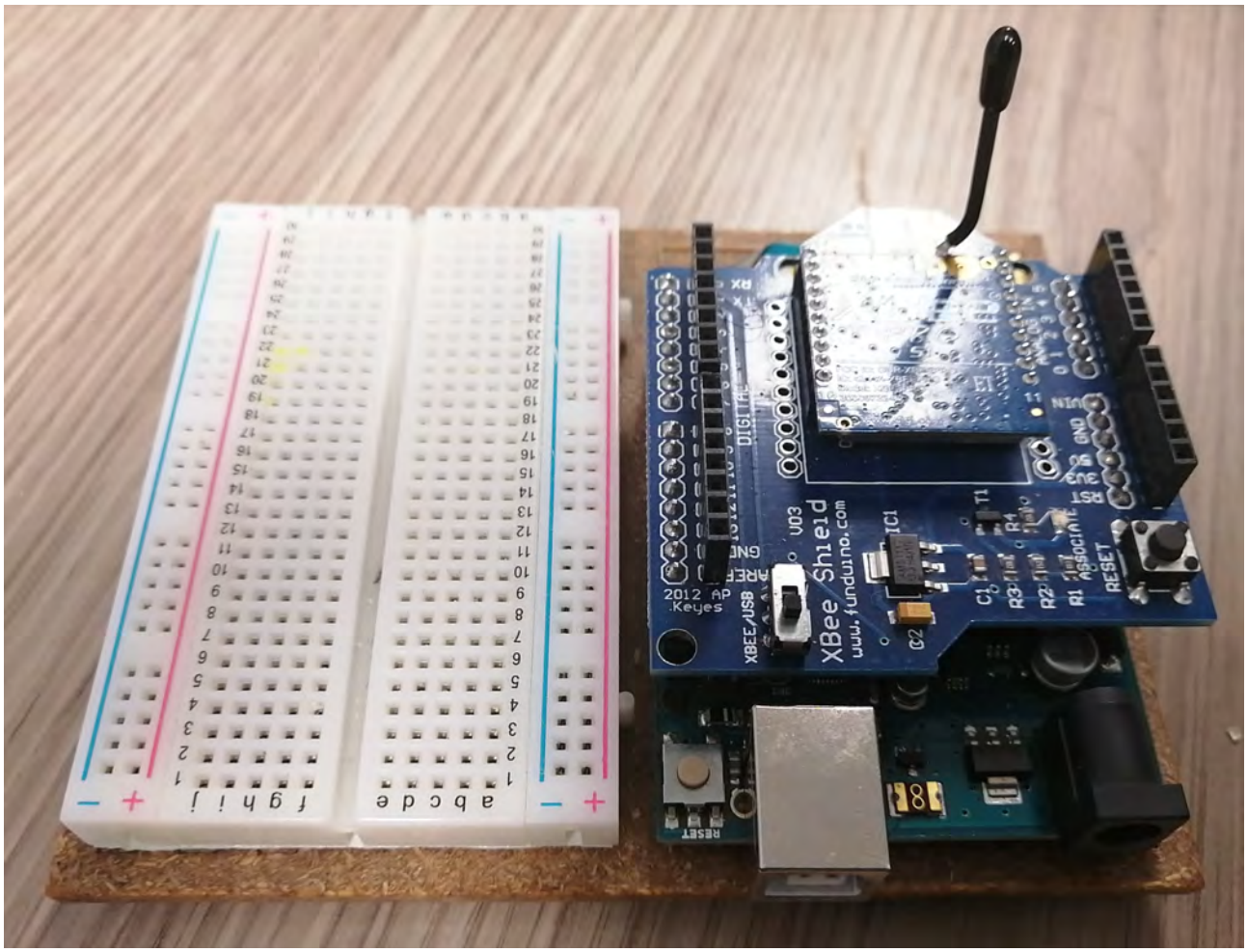

Figure 4.3: Arduino-XBee connection using the XBee shield

4.2.4 Adding our RF Modules to the XCTU Application

As I said earlier, I configured the XBee modules using the XCTU. Before I could configure the XBee modules using the XCTU application, I needed to add them to the application. To do this, I clicked the search icon on the top left corner, and a list of USB ports that are active on the computer were displayed. I then selected the USB ports in which the modules were connected. On the next screen I selected the following default serial port settings:

- Baudrate : 9600
- Data Bits ; 8

- Parity ; None
- Stop bits ; 0
- Flow control : None

The XCTU then scans the selected USB ports, and returns a list of XBee radios their unique 64 bit addresses. I then selected the devices and clicked "Add selected devices".

4.2.5 Configuring our RF Modules

I started experimenting with the XBee series 1 modules. Series 1 XBee modules are based on the 802.15.4 protocol, and supports point-to-point and start network topologies. I configured Point-to-Point network with the selected XBee devices, with a coordinator to start up the network and an end-device. To configure the network, the following parameters as illustrated in Table 4.1: Personal Area Network (PAN) ID, MY Address, Destination Address, and Enable coordinator.

- PAN ID (ID)
  This is the network unique address and it needs to be the same in all the nodes in the network.
- MY Address (MY)
  It is the XBee device 16-bit address, acting as the device unique identifier within the network.
- Destination Address (DH)
  Specifies the XBee modules that the source talks to. This is set by specifying two values, the Destination High Address (DH) and the Destination Low Address (DL). The values can be used in one of two ways, either setting the DH to 0 and matching the source DL value to the destination MY address value. Alternatively, the source DH and DL values can be set to match the destination Serial Number High (SH) and Serial Number Low

(SL) values, respectively. However, for the scope of our experiments, I used the first option.

- Enable coordinator (CE)

  Specifies the role of the node in the network, coordinator or end device

Point-to-Point (P2P)

To configure the two XBee modules in a P2P network such that they are able transmit and receive messages each other, both nodes , 1 and 2, are in the same PAN ID. I set node 1 and 2 MY address as 0 and 1, respectively. I set the nodes DH, the destination high address to 0. This means that the modules communicate on a total of 32 bits, 16 bits for the high address and another 16 bits for the low address. I then set node 1 MY and DL values to 0 and 1 respectively. In addition I set node 2 MY and DL values to 1 and 0 respectively. Node 1 was enabled as coordinator while node 2 was left as an end-device as illustrated in Table 4.1.

	Node 1	**Node 2**
ID	3332	3332
MY	0	1
DH	0	0
DL	1	0
Coordinator Enable	1	0

Table 4.1: P2P network nodes configuration

I then used the XCTU serial console tool to test the transmission between the configured XBee radio nodes. The console communicates directly with the XBee module serial port. As illustrated in Figure 4.4, where the blue text in the left window is the transmitted message, and the red text a received message. I sent "Hello there!" from node 1 to node 2. The messages was successfully transmitted to the node 2 as shown in Figure 4.4. In addition, I successfully transmitted "Hey Hey :)" from node 2 to node 1 as also shown i Figure 4.4.

```
void setup() {
  Serial.begin (9600);
}
void loop() {
  Serial.write(Hello from node 1);
  Serial.read();
  delay (500);
  }
```

Listing 4.6: Testing P2P network: Message from node 1 to 2

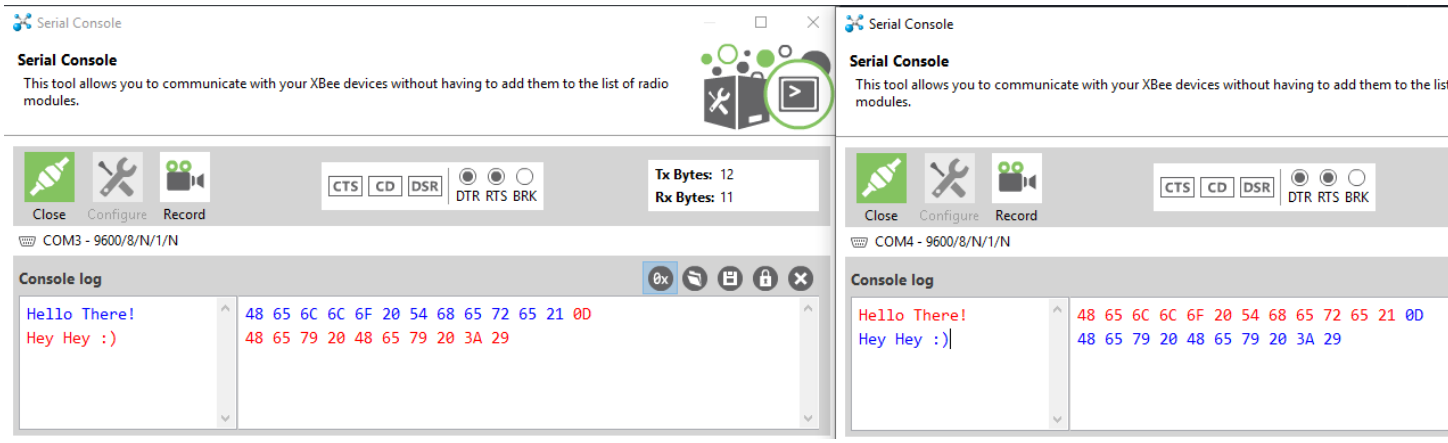

Figure 4.4: Message transmission between node 1 and node 2

Once I had successfully test the transmission between the Arduinos to which the XBee radios were connected. To do this, I wrote and uploaded an Arduino sketch. As illustrated in Listing 4.6, the sketch uses the serial object write data to the Arduino serial port, and then read from it. I used the *Serial.begin* to initialize the serial port in the *setup* function, setting the transmission speed to 9600 bits per second (baud). I then used the *Serial.write* function to write the message to the serial port, and the *Serial.read* function to read incoming data. The messages were indeed successfully transmitted between the Arduino boards.

Star Topology

Once I had successfully tested the P2P network, I proceeded to the test the star network. A star network typically consists of a central node that is able to communicate with multiple nodes, and vice versa. To test the network, I added a third node. I configured the nodes so that node 1 is a coordinator that is able to broadcast messages to node 2 and 3, and that node 1 and 2 and both able to talk to node 1. To achieve this, I set node 1 DH parameter to FFFF, which enables the node to broadcast messages to all nodes in the network as shown in Table 4.2. Following this, I set DH parameter for node 2 and 3 to 0, which is node 1 unique address (MY parameter), which enables both nodes to be able to talk to node 1.

	Node 1	**Node 2**	**Node 3**
ID	3332	3332	3332
MY	0	1	2
DH	0	0	0
DL	FFFF	0	0
Coordinator Enable	1	0	0

Table 4.2: Star network nodes configuration

Mesh Network

The series 1 XBee radio modules that I had used in earlier configurations to implement the point-to-point and star network topologies did not support a mesh network. Series 1 XBee radio modules implement the 802.15.4 stack, which does not support a mesh network. As a result, I switched from the series 1 to series 2 XBee radios (S2C). XBee S2C radios implement the full ZigBee stack, which supports a ZigBee mesh network. ZigBee is a standards-based wireless technology that was developed to allow low-cost, low-power wireless machine-to-machine (M2M) communication and internet of things (IoT) networks [2]. The nodes in a ZigBee network follow a hierarchical structure, with varying computation power and capabilities. [2] mentions three types of ZigBee nodes: namely, an end-device, router and coordinator. Any

ZigBee network requires that the coordinator starts up the network, selecting the frequency to use. Once the network coordinator is up and running, other nodes can join the network simply by specifying the network PAN address and the role the node plays in the network, router or end-device [51]. This makes the coordinator the parent to these devices. In addition, the coordinator may run additional services such as security and routing services, provided that there are no nodes in the network dedicated to run these services [35]. Routers are common in ZigBee networks, but they are not compulsory. They act as intermediate nodes, forwarding messages between nodes. In addition, other devices may join the network through them [35]. End devices are simple nodes which only send and receive messages, and perform no other services [51].

An important ZigBee feature is the ability to support mesh networking. Mesh networks enable any device to communicate with any other nodes in the network, allowing redundant routes for a given node when possible [2]. The ability to forward messages between the nodes enable data to hop between nodes up until it reaches the destination [51]. So, if A and B are out of range from each other, messages can still be transmitted between the two nodes via a node C which acts as intermediate node, provided that node C is within range of both node A and B. In addition, if node C becomes unavailable due for example to a failure, a new route can be automatically established between node A and B provided that there is node D which is within range for both node A and B. [51]. So ZigBee networks are self-healing.

I used three XBee S2C radios, and configured them such that one was the network coordinator (node 1), the second and the third ones were routers as illustrated in Table 4.3. To do this put all nodes under the same PAN ID address. I then enabled the coordinator parameter in node 1, which makes XBee radio a coordinator in the network. I also enabled the channel verification parameter for the routers. This parameter verifies if there is a coordinator in the channel after the router joins, if there is none found, then the router exits the channel.

Once the XBee radios were configured in a ZigBee mesh network, I proceeded to check the message transmission between the nodes. Again, I used the XCTU serial console to test this communication. I transmitted the message "Hello1" and "Hello2" from router 1 and

	Coordinator (Node 1)	**Router 1** (Node 2)	**Router 2** (Node 3)
PAN ID	3332	3332	3332
Destination Address	-	0	0
Coordinator Enable	1	0	0
Channel Verification	0	1	1

Table 4.3: ZigBee mesh network nodes configuration

2 respectively to the coordinator. In addition, I sent the message "Hello there" from the coordinator node which was then broadcasted to the router nodes.

As seen in all these experiments, I used the XCTU software to configure and test the XBee modules. The main reason for this is that the software simplifies the configuration of the modules. However, the XCTU software would not be practical for implementing larger networks. For example, if the network nodes have already been deployed in the field, to reconfigure them, one would have to remove them on the field and connect them to a PC running the XCTU. This would make more difficult the overall implementation and management of the system. Alternatively, I could have used AT commands to configure the XBee programmatically. To do this, I would write the XBee module settings in the Arduino setup function such that the modules are configured when the Arduino board is powered.

4.3 Conclusion

This chapter first reported on the set of experiments to learn to work with the Arduino, i.e. building electric circuits, and writing Arduino programs to control the circuit elements. During

these experiments, building the electric circuits was a completely new task for me, however, I gained experience through available guides to build electric circuits. When it came to programming the Arduino, my strong programming background as a Computer Science post graduate gave me an advantage although I was new to the Arduino IDE programming language. Secondly, this chapter explored how XBee devices are configured, and the different network topologies supported by XBee devices, transmitting messages directly among XBee devices, and then among the Arduinos integrated to the radios. During this experiment, I used the XCTU platform to configure the XBee devices, and also test their direct communication among each other. The platform allows to interact with XBee devices in a very user friendly graphical interface. In addition, there was a great amount of support available online to teach one to configure XBee modules on XCTU in all the different network topologies. This made configuring and testing the XBee devices a very easy task. When it came to integrate the XBee devices into the Arduinos, there was also a great amount of support available online, which made this an easy task. Also, testing the communication wireless communication between Arduino boards using the XBee devices was an easy task, mainly because of the great support available online, and also that, at this point I had been acquainted with the Arduino IDE language.

Chapter 5

Communication between Arduino and Raspberry Pi Over XBee

This chapter reports first on an experiments to illustrate a wireless communication between an Arduino and the Raspberry Pi using the XBee devices in "transparent mode" (essentially building a wireless serial interface). A second experiment explores the advanced XBee API mode, highlighting significant differences between the API and "transparent" modes. The XBee devices are reconfigured in API mode, and used for a more sophisticated wireless communication between the Raspberry Pi and the Arduinos.

5.1 Connecting Arduino to Raspberry Pi Using XBee

5.1.1 Aim

The aim of this experiment is to understand how to get the Arduino boards to communicate with the Raspberry Pi board using XBee radios.

5.1.2 Tools

- Arduino Uno board

- Arduino XBee shield
- 2x XBee S2C modules
- 1 x Raspberry Pi 3
- 1 x Raspberry Pi XBee shield
- Power Supply 3 x (USB A-B cable)
- Windows PC
- Arduino IDE

5.1.3 Booting the Raspberry Pi

Raspbian Image

Before I could use the Raspberry Pi board, I first had to load an operating system. To do this,

- ✓ the Raspberry Pi SD card was formatted as a FAT file system.
- ✓ I then downloaded the latest version of the Raspbian into the SD card from the : `http://www.raspberrypi.org/downloads/`
- ✓ Following this, I used the Win32 Image writer (from `https://launchpad.net/win32-image-writer`) to write the image into the SD card, to have a bootable SD card.
- ✓ I inserted the SD card in the Raspberry Pi SD card slot and powered the Raspberry Pi
- ✓ At this point, the Raspberry booted from the SD card. I then fixed the general settings such as timezone, locale, etc.

5.1.4 Configuring the Serial Port

I then proceeded setting up the Raspberry Pi UART serial port on the UART transmit and receive pins GPIO 14 and 15, which the board will communicate with the XBee module. This was done in the following steps:

- ✓ I first enabled UART on the Raspberry Pi
  - Run the command line
  - Open the "/boot/config.txt" file using the the command ***sudo nano /boot/config.txt***
  - Add the line _uart=1 at the end of the file to enable the UART serial port on the Raspberry Pi
  - Restart the Raspberry Pi using the ***sudo reboot*** command
- ✓ Once the UART port (S0) was enabled on the Raspberry Pi, I proceeded to remove console service. This is due to the fact that, by default, the Pi's UART is used for console service, and so to use the UART port with serial devices, I need to first disable the console service. To do this, I removed the line "console=serial0,115200" from the "/boot/cmdline.txt" file.
- ✓ I then exited the editor and Rebooted the Raspberry Pi
  - ***sudo reboot***

By default, the Raspberry Pi ttyAMA0 serial port is connected to the Bluetooth function. And so, to free the serial port for our own use, the miniuart-bt UART Tree Overlay was used. With this, the Raspberry Pi 3 Bluetooth function is switched such that, instead of using the UART0/ttyAMA0 uart port, uses the ttyS0. In addition, the UART0/ttyAMA0 is restored to GPIOs 14 and 15. To do this, I added the line "dtoverlay =pi3-miniuart-bt" to the "/boot/config.txt" file. At this point, I had our Raspberry Pi UART port setup and accessible at /dev/ttyAMA0.

5.1.5 Arduino - Raspberry Pi Communication

I then tested the wireless communication between Arduino boards and the Raspberry Pi as shown in Figure 5.1.

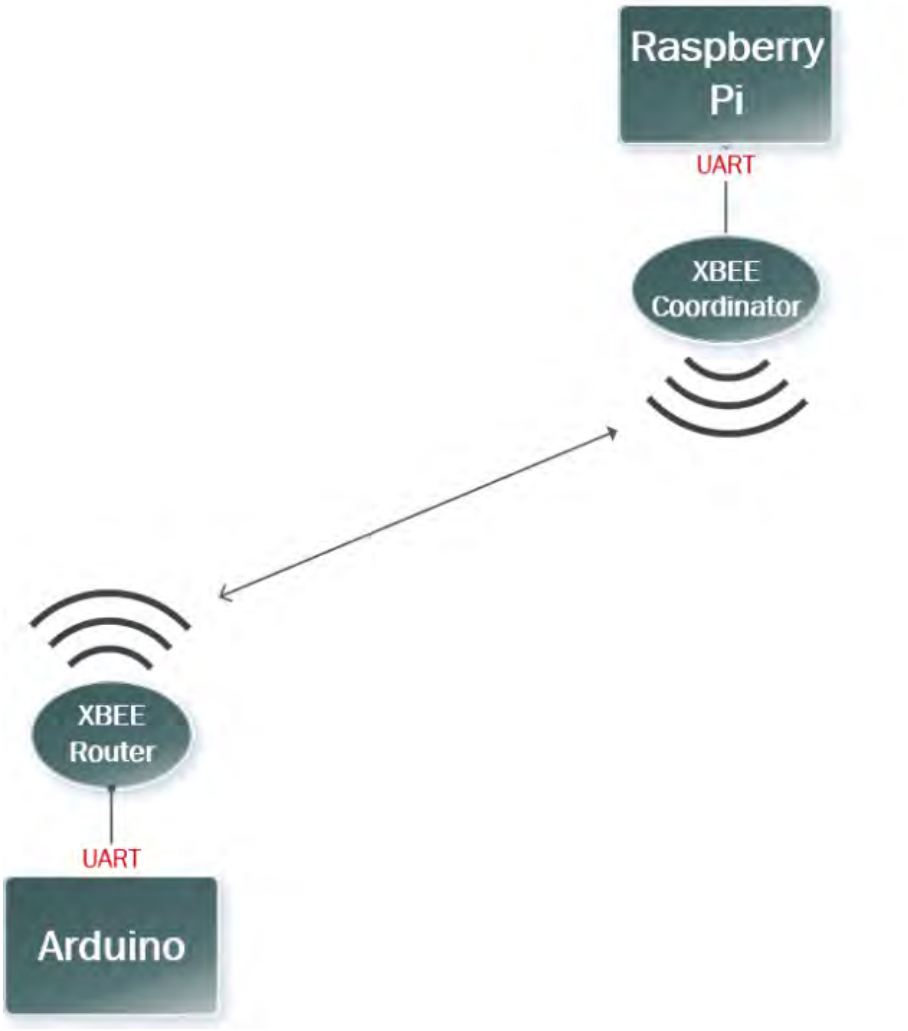

Figure 5.1: An illustration of the setup to test the communication between Arduino and Raspberry Pi

I powered all units, the Raspberry Pi with an external power supplier, and the two Arduino boards using a USB cable connected to a Windows PC. I then used the Arduino IDE to write and upload simple programs that send a message " Hello 1" from the Arduino to the Raspberry Pi as illustrated in Listing 5.1. The program uses the *Serial.begin* function to open a serial port, and then uses the *Serial.println* function to write to the serial port the

```
void setup() {
  Serial.begin(9600);
}
void loop() {
    Serial.println("Hello 1");
    string incoming="";
    incoming = Serial.readStringUntil();
    Serial.println("Recieved from Pi:" + incoming);
    delay(5000);
  }
```

Listing 5.1: Arduino code to send and receive data from the Raspberry Pi (adapted from [16])

message "Hello1". I then placed the USB switch on the "XBee" position so that the PC could communicate to the Arduino micro-controller. I uploaded the sketch to their respective Arduino boards.

I then plugged the a configured XBee router from Experiment 2 into the Arduino board using the Arduino XBee shield. In addition, I plugged the configured XBee coordinator into the Raspberry Pi shield.

To check if the messages were transmitted to the Raspberry Pi, I wrote a Python script that reads and displays data on the Raspberry Pi serial port as illustrated in Listing 5.2.

The code imports the Python serial library. A Raspberry Pi Serial instance is then configured and open. I then used I then used a Python while loop to infinitely read and display data from the serial port. The messages were successfully transmitted from the Arduino to the Raspberry Pi, and displayed on the Raspberry Pi screen.

I then added a second Arduino, connected with another configured XBee router using

```
import serial
ser= serial.Serial (
                    port= '/dev/ttyAMA0',
                    baudrate = 9600)
ser.close()
ser.open()
while True:
    incoming = ser.readline().strip()
    print (incoming)
```

Listing 5.2: Python code for displaying data received on the Raspberry Pi (adapted from [70])

the XBee shield. I then transmitted a similar message "Hello 2" to the Raspberry Pi. The messages were successfully transmitted to the Raspberry Pi and displayed on the Raspberry Pi screen.

5.2 XBee API Mode

XBee devices support two types of sending and receiving data from and to controllers via their serial port, namely, the Transparent (AT) and the Application Programming Interface (API) mode. I had explored the AT mode with previous experiments because, by default XBee modules are configured to work in AT mode.

5.2.1 AT Mode

With the AT mode, the XBee module acts as a serial line replacement. Any data that is received by the module through its serial input is immediately transmitted as it is by the radio. In addition, data that is received by the radio is forwarded through the serial interface.
For XBee modules to communicate, the sending module must know who else to send data to. This is specified during the module configuration (i.e the DH and DL parameters) as mentioned

in previous experiments. The AT mode is typically used for simple point-to-point links, where it offers easy operation, useful for getting started with XBee devices. However, there are a number of limitations about the AT mode. Data can only be sent to one address which is predefined, to transmit data to a new address, the XBee module configuration needs to be updated. In addition, when data is received, the is no indication or details of the sender. Most importantly, it does not directly support mesh networks, a requirement for our system.

5.2.2 API Mode

In this mode, data that is communicated through the serial interface is organised in packets. In line with what I have said in Section 5.2.1, the advantages of using the API mode are as follows:

- The API mode offers different types of frames such as configuration and communication frames, allowing device configuration without entering command mode
- On the sending side API frames include destination (i.e. destination address) which enables to send data to multiple devices without the need to reconfigure the XBee device
- Received API frames carry details about the sender, thus making it easy to trace who sent the packet.
- ZigBee mesh capabilities are directly and easily supported.

The advantages to the API mode minimize the limitations to using the AT mode. However, a downfall to using the API mode is that the interface is more complex as the packets are organised in a specific format.

5.2.3 API Mode Packets

As mentioned in Section 5.2.2, the API mode offers different types of packet frames, with are either sent to or received from remote XBee modules through the serial interface. Examples of the frames sent by XBee modules include, AT Command and Transmit Request. Examples

of frames received by an XBee modules include an AT Response, Transmit Status and Receive Packet. Each frame has its own structure, consisting of numerous parameters indicating the operation that should take place. For the scope of this experiment, I focused on the Transmit and Receive Packets.

5.2.4 Configuring Nodes in API Mode

The XBee modules were reconfigured in API 2 mode using the XCTU. To do this, I changed the XBee module API parameter. The API parameter may hold one of three values, namely, the AT, API 1 or API 2. The difference between API 1 and API 2 is that API 2 requires escape characters, to improve reliability in noisy environments.

	Coordinator (Node 1)	**Router 1** (Node 2)	**Router 2** (Node 3)
PAN ID	3332	3332	3332
Destination Address	FFFF	0	0
Enable API	2	2	2
Coordinator Enable	1	0	0
Channel Verification	0	1	1

5.2.5 Data Transmission from Module A to B: XCTU

I tested the communication between the nodes in the XCTU environment using the software Serial Console. The Serial Console consists of the Frames Log, the Frame details and the Sending Frame sections. The Frame log is where the incoming and outgoing frames are displayed in red and blue, respectively. The Frame details contains details about a specific frame such as the frame source and destination details. The Sending Frame is where API frames are generated

using the frame generator tool.

As illustrated in Figure 5.2, I created an API Transmit Request frame using the API frame generator Tool. The structure of a transmit frame is as follows:

- Start Delimiter
- Frame Length
- Frame data
- Checksum

The Start delimiter indicates beginning of a new frame and is by default *OX7E*. The frame length is the number of bytes between the frame length and the checksum. The frame data contains the information the frame. The type of frame is a Transmit Request (0x10) in my case. In addition, the frame data contains the destination address, either the 16 or 64 bit address of the XBee radio module I want to talk to (00 13 A2 00 41 7D E4 39). The RF data is the actual data I want to send in the packet.

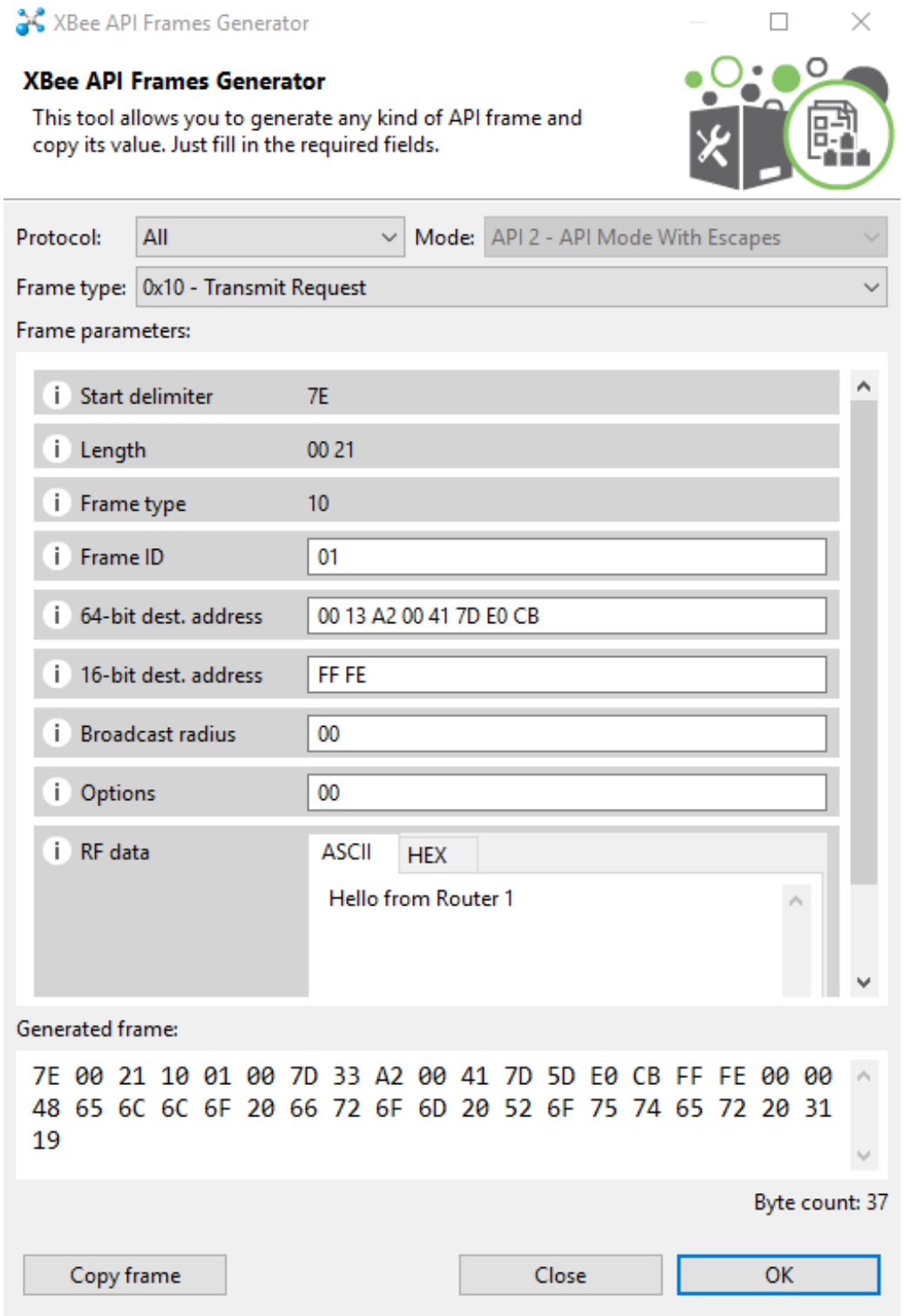

Figure 5.2: Creating a Transmit Request Frame with the XCTU API frame generator tool

I then sent the created frame, and the packet was successfully transmitted through the

serial interface and received in module B as seen in Figure 5.4. In addition, module A, the sender, received an acknowledgement of a successful data transmission as a Transmit Status packet as shown in Figure 5.3.

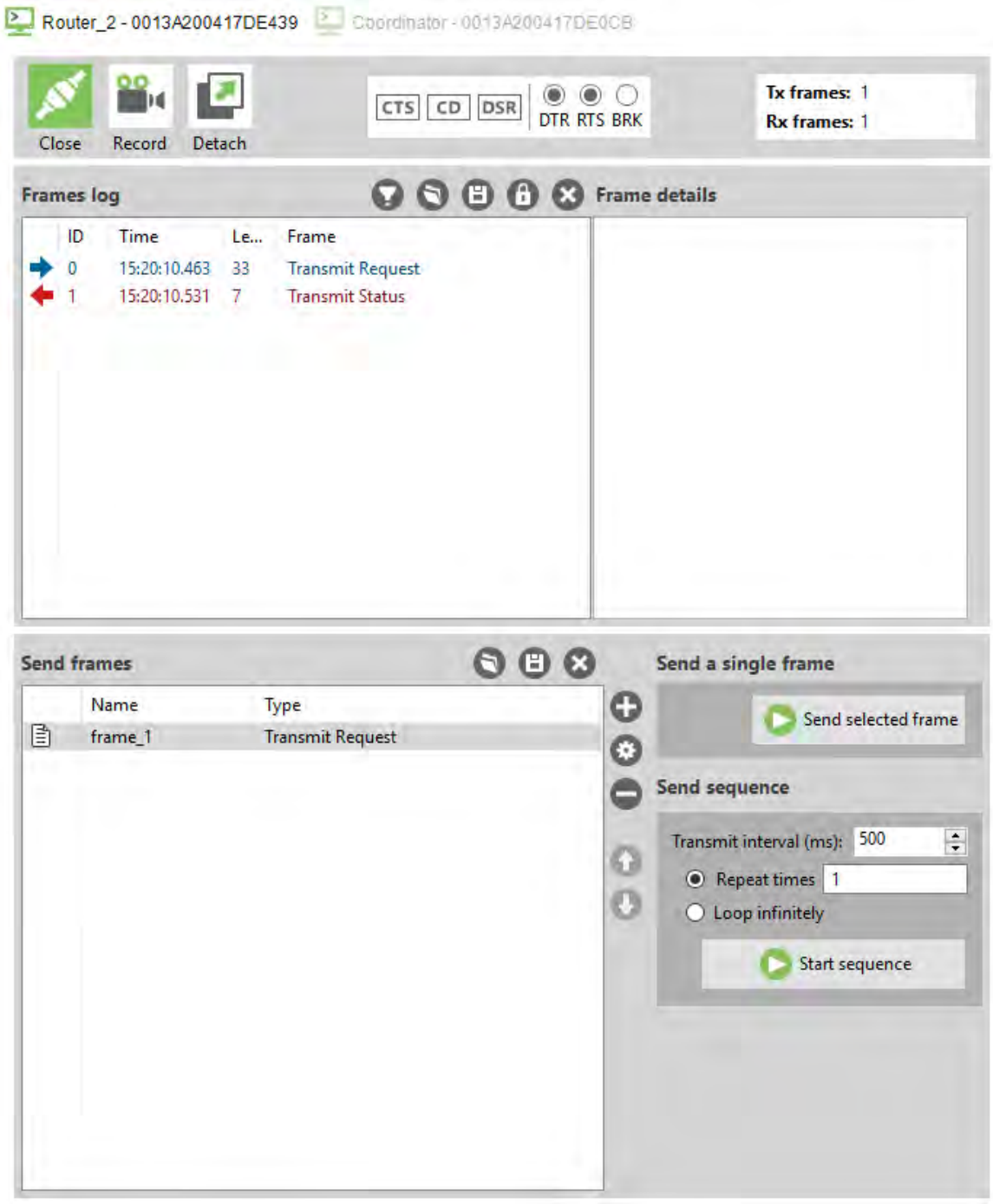

Figure 5.3: Transmission of an API Transmit Request frame from module A

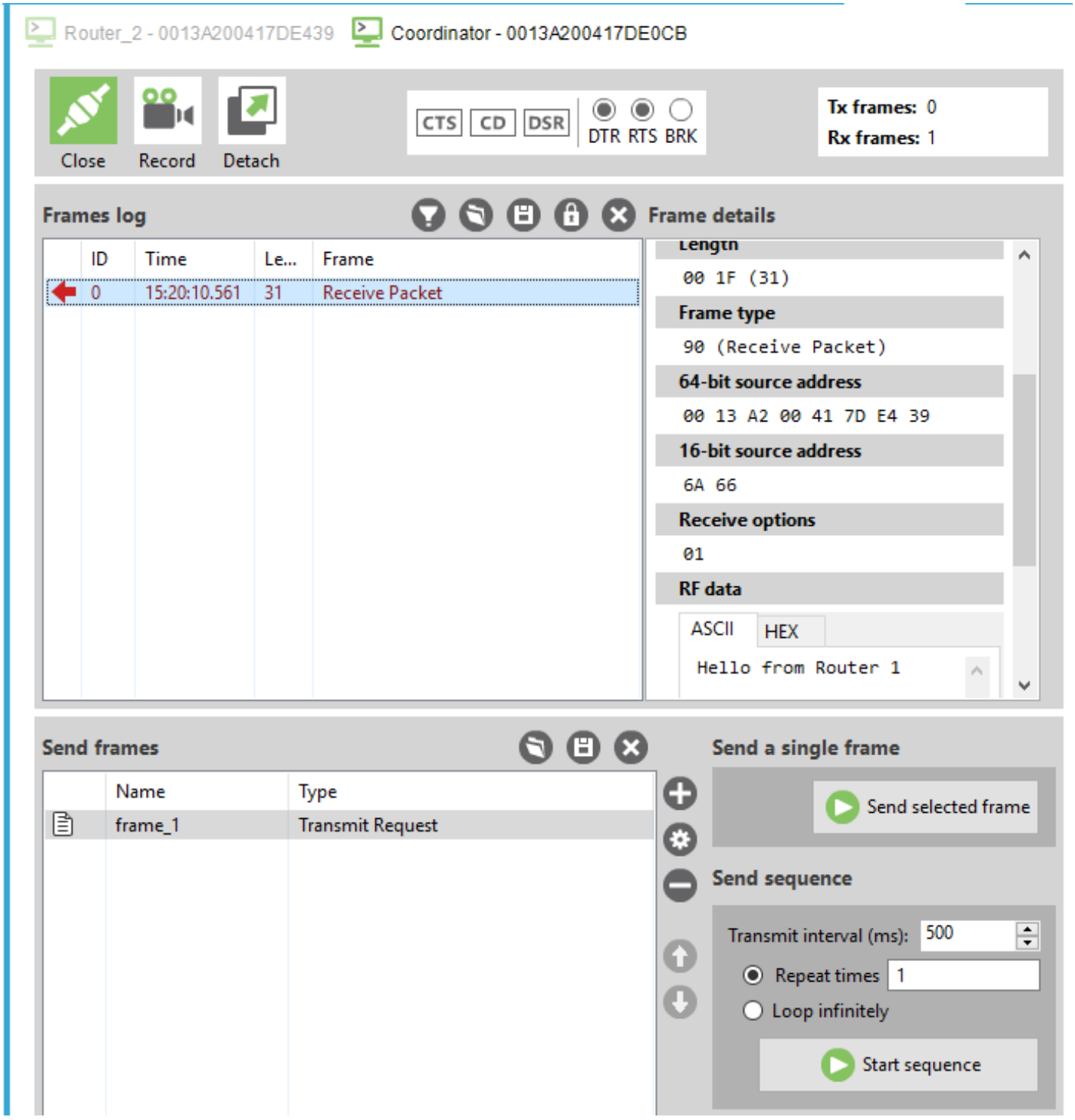

Figure 5.4: Module B received packet

Following the successful transmission of data from module A to B, I then tested data transmission from module B to A. Similarly to the previous testing, i created an API Transmit

Request frame with the XCTU API Frame Generator Tool, specifying the destination address (0013A200417DE439) and the RF data. I then sent the the created frame, which was then transmitted through the serial interface and received by module A. In addition, module A received an acknowledgement of a successful data transmission as a Transmit Status packet.

5.2.6 Data Transmission from Module A to B: Arduino and Raspberry Pi

At this point, I was sure that the XBee modules were able to communicate and exchange packets through in API mode. Following this, I tested the data transmission from an Arduino board to the Raspberry Pi board via XBee in API mode.

Data Transmission from the Arduino

In this experiment, I used the XBee-arduino library for communicating with XBee radio modules in API mode. The library naturally allows the XBees to interact my Arduino programs. The library supports a majority of packet types, including Tx/Rx packets. As shown in Listing 5.3 , I first imported library into the Arduino sketch, and then created an XBee object. The object consists of functions for sending and receiving packets. I then opened a serial port, and then set the XBee object to communicate through the open serial port, using the *SetSerial()* function. Following this, I created a *ZBTxRequest()* object, which represents a ZigBee Transmit Request API frames, specifying the following parameters:

- The actual data I want to transmit in the packet
- Destination address
- Size of the data I are transmitting

The Tx Request was then sent using the *XBee.send()* method.

```
//0013A200417DE439 *Arduino
//0013A200417DE0CB *RaspberryPi
//0013A200417DE0BA *Second Router

#include <XBee.h>
XBee xbee = XBee();
uint8_t text1[] = {'H', 'e', 'l', 'l', 'o', 'P','i'};

XBeeAddress64 remoteAddress1 =  XBeeAddress64(0x0013A200, 0x417DE0CB
    ); //pi
ZBTxRequest zbTx1 = ZBTxRequest(remoteAddress1, text1, sizeof(text1)
    );

void setup () {
  Serial.begin(9600);
  xbee.setSerial(Serial);
}

void loop () {
    xbee.send(zbTx1);
    delay(10000);
}
```

Listing 5.3: Arduino program to transmit data to the Raspberry Pi (adapted from [5])

Data Collection from the Raspberry Pi

It would have been a difficult task to interact with the XBee modules using the XCTU. This is due to the fact that the Raspberry Pi runs on Raspbian OS, an operating system to which the XCTU has not yet been ported. Therefore, to download the XCTU on the Raspberry Pi, I would have to put a Windows emulator on the Raspberry Pi. I used instead the Python-XBee library on the Raspberry Pi. The library allows to communicate with XBee modules in the Python environment. I installed the library from Github (git clone git@github.com:digidotcom/xbee-Python.git). Once the installation was finished, I then wrote a Python script to communicate with the XBee modules which is illustrated in Listing 5.4. In the program, I first imported the *XBeeDevice* class. The class object represents the physical XBee module in the API mode. A physical XBee modules runs specific wireless communication protocols, i.e ZigBee device, 802.15.4 device, depending on the module hardware and firmware. For this reason, the *XBeeDevice* also supports protocol-specific objects. The classes allow the configuration of the physical XBee device, communication with other devices (sending and receiving data). The classes may have additional protocol-specific features depending on the class.

The class objects can be instantiated as either local or remote devices. A local device is one that is physically attached to the PC through a serial port. On the other hand, remote devices are not physically attached to the PC, but they are reachable through the network to which the attached local device belongs. For this experiment, I instantiated the XBee device object as a local object since it XBee device is physically attached to the Raspberry Pi through the serial port. In addition, I instantiated the object specifying the serial port details, */dev/ttyAMA0*.

I then opened the object serial port connection using the *open()* function. I created an infinite loop to read and print API packets that were received on the serial port. Inside the loop I used the *read_data()*. The method takes an integer, the timeout, and blocks application until data from any XBee device of the network is received or the timeout provided has expired and returns the data inside the XBee message. The received data contains the following retrievable information:

- The RemoteXBeeDevice that sent the message.
- Byte array with the contents of the received data.
- Flag indicating if the data was sent via broadcast.
- Time when the message was received.

However, it returns none if no data was read. I used the *data.decode()* and the *remote_device.get_64bit_addr()* XBee message method to decode the frame data and retrieve the address of the frame sender, respectively. The received frame data was displayed on the Raspberry Pi screen, as well as the sender XBee radio device address. Following this I closed the serial port connection.

5.2.7 Data Transmission from Raspberry Pi to the Arduino

After I tested the data transmission from the Arduino to the Raspberry Pi, I proceeded to testing the data transmission from the Raspberry Pi to the Arduino.

Data Transmission from the Raspberry Pi

Still using the created XBee device object, I added a few lines in our Python program to handle the data transmission. I used the *send_data* method to transmit data from our XBee device object to the remote XBee device that sent the message as seen in Listing 5.4. The packet was received on the Arduino.

```
from digi.xbee.devices import XBeeDevice \\
device = XBeeDevice("/dev/ttyAMA0", 9600) \\



def main():
    try:
        device = XBeeDevice("/dev/ttyAMA0", 9600)
        device.close()
        device.open()
        while True:
            mess_data = device.read\_data()
            sender\_addr=message.remote_device.get_64bit\_addr()


                print ("Read %s from %s" %
                       (mess\_data,
                        sender\_addr))
            if message is not None:
                 print ("Read %s from %s" %
                       (mess\_data,
                        sender\_addr))
    finally:
          if device is not None and device.is\_open():
              device.close()


if __name__ == '__main__':
    main()
```

Listing 5.4: Python script to handle data collection on the coordinator (adapted from [50])

5.3 Conclusion

This chapter reported on the set of experiments I conducted to learn to work with a Raspberry Pi and how to get the Raspberry Pi to communicate wirelessly with an Arduino using XBee devices. While getting started with the Raspberry Pi, I first loaded an operating system and proceeded to configure the serial port which I would use later to interact with XBee devices on the Raspberry Pi. During this task, I relied solely on the support available online, which was very detailed and easy to follow. Following this, I proceeded to test the wireless communication between the Raspberry Pi and an Arduino, using the configured XBee devices. During this task, I also explored the two ways in which XBee devices send and receive data to and from controllers, the Transparent and API mode, where the latter transmits data in packets and fully supports ZigBee mesh networks, and was therefore more suitable for the smart irrigation system that I was trying to build. I then tested the wireless communication between the Raspberry Pi and the Arduino using the XBee devices in API mode. Because the API mode interface is more complex than AT mode, I made use of available support and tutorials online.

Chapter 6

Experimenting with MQTT and MQTT-SN

This chapter reports on the experiments I conducted with MQTT protocol. First, the chapter briefly explains the protocol, and then records MQTT experiments I conducted on the Raspberry Pi using the command line clients provided by the Mosquitto project. Following this, the chapter records the experiments I conducted with Python MQTT clients on the Raspberry Pi. Finally, the chapter explores the implementations of the MQTT-SN protocol, a super-set of MQTT supporting ZigBee and UDP.

6.1 Experimenting with MQTT

Message Queuing Telemetry Transport (MQTT) is an IoT data transfer protocol running on top of TCP/IP stack. MQTT was designed as a lightweight messaging protocol in order to suit connections with remote locations that require a "small code footprint" or limited bandwidth, based in the subscriber/publisher massaging pattern [18]. An important feature about this subscriber/publisher messaging pattern is that it decouples the subscribers from the publishers [96]. The subscribers act independently from the publishers and vice versa, and establishes an asynchronous communication between the publishers and subscribers [18]. The communi-

cation of the subscribers and publishers with the server is on the other hand, synchronous. Another important feature about the publish/subscriber messaging pattern is that one publisher can send data to different subscribers. Also, because the data is published to the server, the published data can be accessed by multiple subscribers without the need for the publisher to explicitly send the data to each and every subscriber [18]. Similarly, one subscriber can access data from multiple publishers [96].

An MQTT client is either a subscriber, publisher, or both. An MQTT client can be one or both. Clients connect to the broker and they can either publish or subscribe to a topic in the broker. Publishers publish or send data on a topic to the MQTT broker. Subscribers retrieve data from the broker. When a publishers publishes a message on a topic, the broker forwards the message to any clients that subscribed to the topic.

6.1.1 Testing MQTT Clients on Raspberry Pi Command Line

For the experiments, I used the open source Eclipse Mosquitto message broker. The broker implements MQTT version 5.0, 3.11 and 3.1 [26]. In addition, the broker provides a C library to implement MQTT clients, as well as command line clients, *mosquitto_sub* and *mosquitto_pub* clients [26]. I installed the broker on the Raspberry Pi using: *sudo apt-get install -y mosquitto*. Once the broker was successfully installed, I made Mosquitto autostart on boot using the command line *sudo systemctl enable mosquitto.service*. Then I installed the command line Mosquitto clients, using the command: *sudo apt-get install mosquitto-clients*.

Once I had the MQTT broker and command line clients installed, I proceeded to experiment with MQTT clients publishing and subscribing. I opened a terminal window 1 to run an MQTT subscriber client to subscribe to the topic "myTopic". To do this, I ran the following command: *mosquitto_sub d t myTopic* as shown in Figure 6.1. I created open a second terminal window to publish a message to the topic "myTopic" using the following command: *mosquitto_pub d t myTopic m "Hello there!"* (see Figure 6.2). Once I published a message to the

topic, the message was received and displayed by the subscriber client as shown in Figure 6.1. I added another MQTT subscriber to the same topic "myTopic" on a third terminal window. I published another message "Hello there" to the topic "myTopic" using the MQTT publisher client. The message was received by both subscriber clients as shown in Figure 6.1 and 6.3.

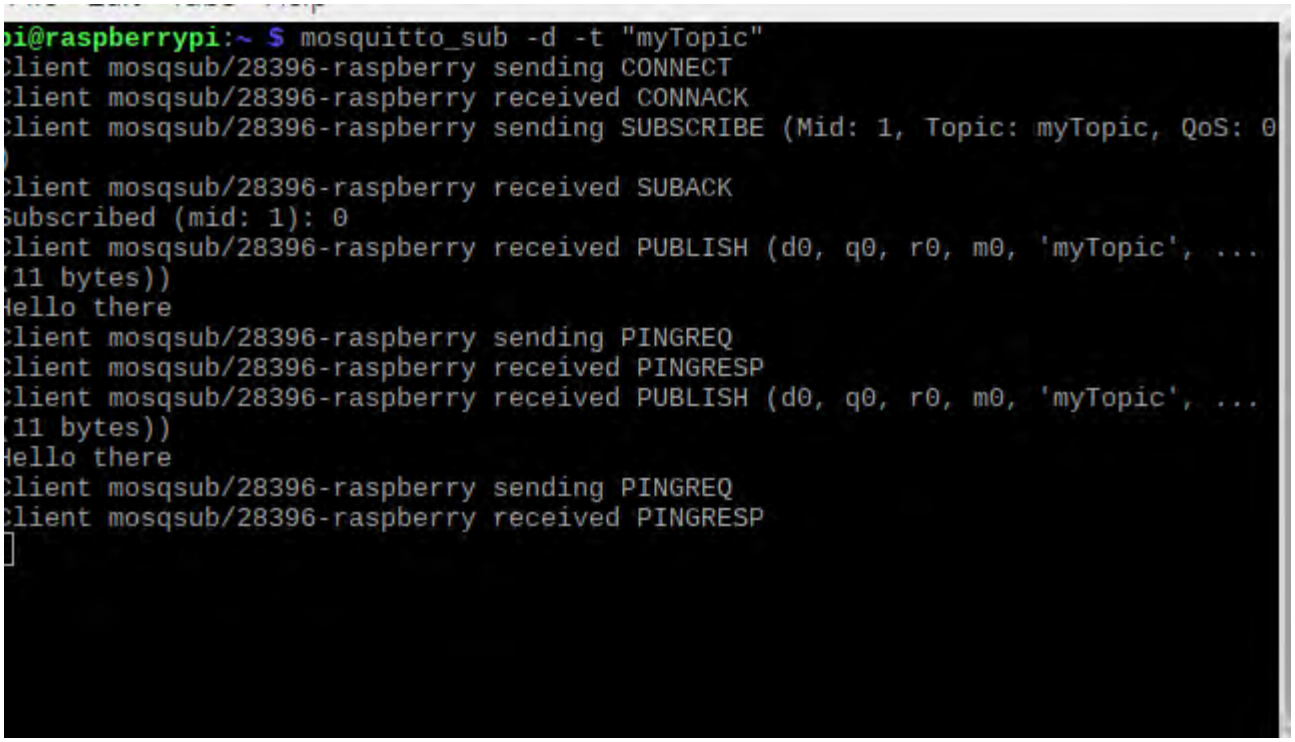

Figure 6.1: Terminal Window 1: MQTT subscriber

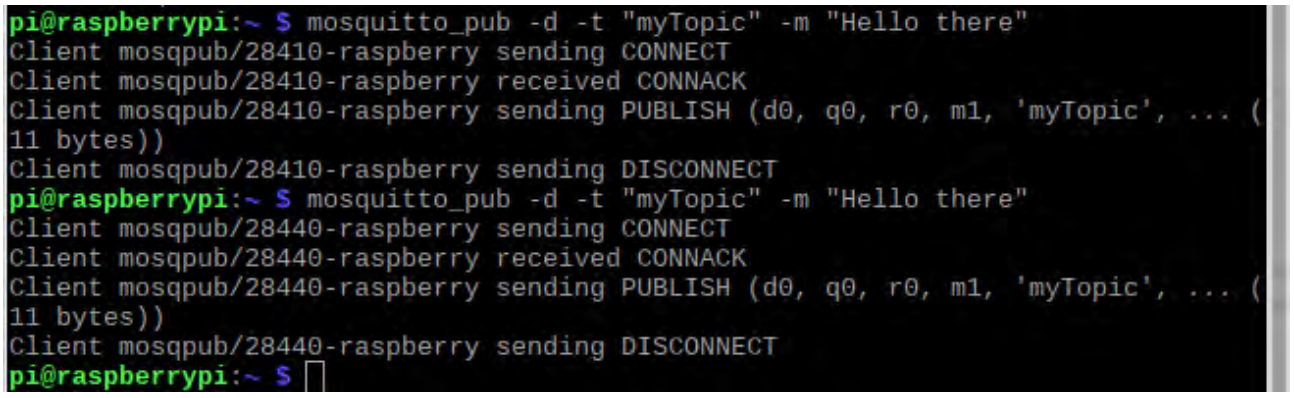

Figure 6.2: Terminal Window 2: MQTT publisher

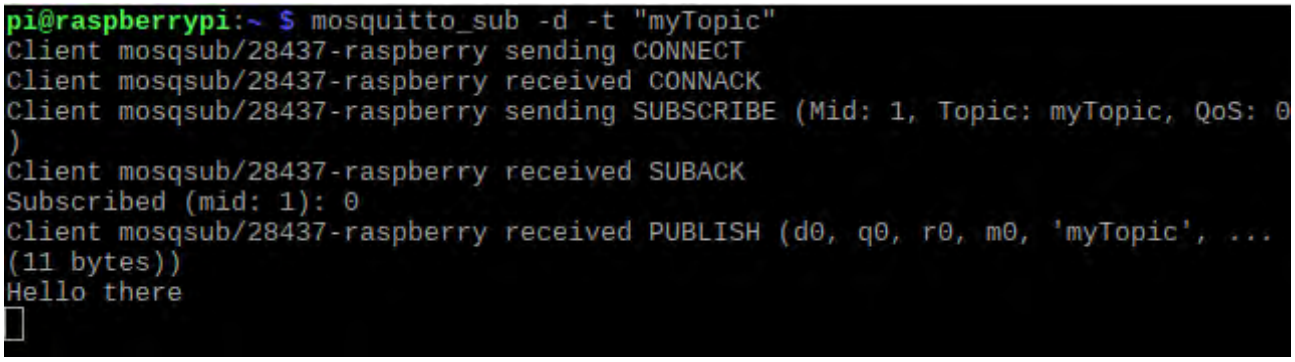

Figure 6.3: Terminal Window 3: MQTT subscriber

6.1.2 Testing MQTT Clients on Raspberry Pi: Python MQTT Client Library

At this point, I had successfully explored MQTT publish/subscribe using the Mosquitto command line clients. I proceeded to explore the Python MQTT client library. The library provides an MQTT client class, enabling applications to connect to an MQTT broker in order to publish to topics and subscribe to topics and retrieve data. The Library implements MQTT version 3.1 and 3.11 [47]. In addition, it fully supports Python 2.7.9+ or 3.4+ and offers limited support to Python 2.7 and Python versions before 3.7.9 [47].

I installed the Python MQTT client library on the Raspberry using the command *pip or pip3 install paho-mqtt*. Following this, I wrote and executed a Python program illustrated in Listing 6.1, to implement the Python MQTT subscriber client. To do this, I first imported the *paho.mqtt.client* class, and initialized an MQTT client object of the class. I connected the client to the Mosquito broker as illustrated in Listing 6.1, and then subscribed the client to the "myTopic" topic.

Following this, I created a second instance of the client class, also connected to the running Mosquitto broker, but running on a different Python program. The client publishes data to the "myTopic" topic as illustrated in Listing 6.2.

```
import paho.mqtt.client as mqtt
client = mqtt.Client()
client.connect("localhost", 1883, 60)
client.subscribe("myTopic")
client.loop_forever()
```

Listing 6.1: Python program implementing a Python MQTT Client Library: Subscriber (adapted from [81])

```
import paho.mqtt.client as mqtt
client = mqtt.Client()
client.connect("localhost", 1883, 60)
client.publish("myTopic", "Hello there")
```

Listing 6.2: Python program implementing a Python MQTT Client Library: Publisher (adapted from [81])

Once the data was published to the topic, the published message was received and displayed by the subscriber client which I created earlier (as seen in Figure 6.4).

```
pi@raspberrypi:~ $ sudo python /home/pi/Documents/mqttclass.py
Connected with result code 0
Received message 'Hello there' on topic 'myTopic' with QoS 0
Received message 'Hello there' on topic 'myTopic' with QoS 0
```

Figure 6.4: Terminal Window 1: An illustration of a Python MQTT client receiving published data through an MQTT subscription

I created another MQTT subscriber client, again using the Python MQTT library. Similar to the subscriber I created earlier, the client was connected to the running Mosquitto broker and subscribed to the "myTopic" topic as well. Following this, I re-executed Python MQTT

publisher program. Once the data was published, the message was received and displayed by both subscriber clients as shown in Figure 6.4 and 6.5.

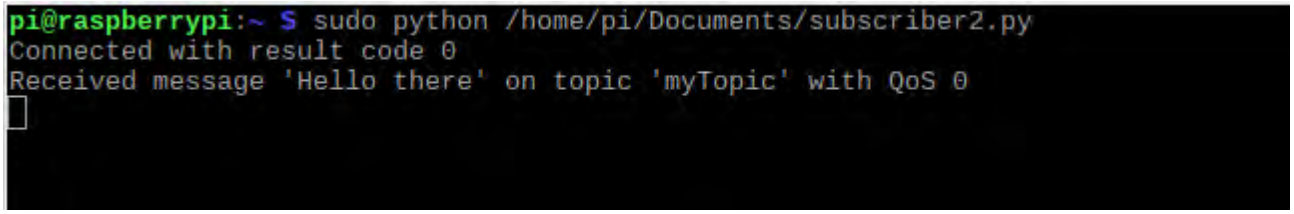

Figure 6.5: Terminal Window 2: An illustration of the second Python MQTT client receiving published data through an MQTT subscription

6.2 Experimenting with MQTT-SN

I wanted to implement MQTT at the level of the sensor network nodes in order to have nodes acting as MQTT clients. However, as mentioned before, MQTT requires the TCP/IP stack and does not support ZigBee. As a solution to this, a superset of the MQTT protocol MQTT-SN was developed [80]. MQTT-SN was designed to operate on low-power and low cost SA devices, adding the possibility of using ZigBee and UDP, while naturally maintaining the MQTT pub/sub model [36][80]. Similarly to MQTT, MQTT-SN architecture consists of MQTT-SN clients (publishers and subscribers), and a standard TCP-based MQTT broker (server). However, MQTT-SN has an additional component, an MQTT-SN gateway as seen in Figure 6.7 [80]. The gateway acts as a protocol conversion between MQTT and MQTT-SN, allowing the MQTT-SN clients connect to the MQTT broker via the MQTT-SN gateway [36]. In addition, the gateway needs to be connected to a broker.

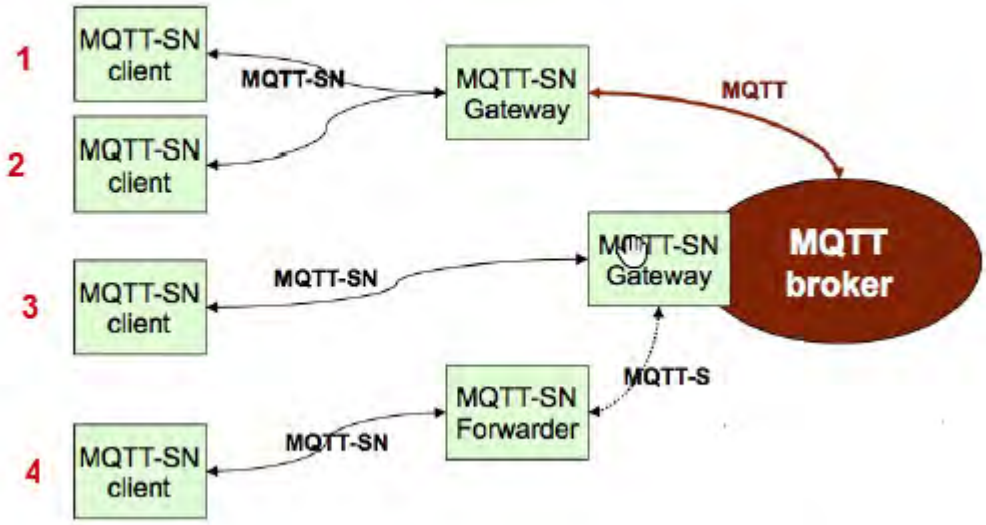

Figure 6.6: MQTT-SN protocol architecture (adapted from [80])

There are two types of MQTT-SN gateways, an aggregating gateway and a transparent gateway. With a transparent gateway, the gateway sets up and maintains an connection to the MQTT broker and performs the protocol translation between the MQTT and MQTT-SN protocols [36]. On the other hand, with the aggregating gateway, the gateway establishes one MQTT connection to the broker for all the MQTT clients and chooses which messages from the clients are transmitted to the MQTT broker [36].

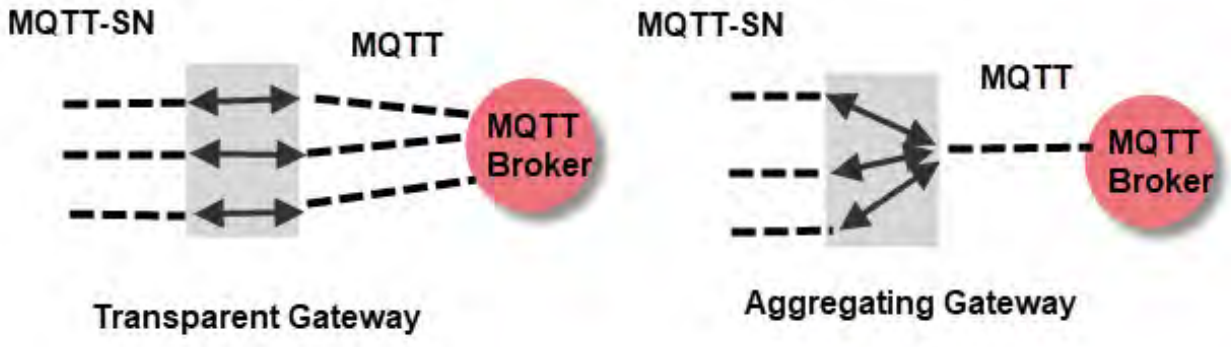

Figure 6.7: MQTT-SN Transparent and Aggregate gateway (taken from [80])

```
$ git clone -b experiment https://github.com/eclipse/paho.mqtt-sn.
 embedded-c
$ cd paho.mqtt-sn.embedded-c/MQTTSNGateway
$ make [SENSORNET={xbee}]
$ make install
$ make clean
```

Listing 6.3: Building an MQTT-SN gateway on the Raspberry Pi (Adapted from [100])

In this architecture, the clients need to be able to discover gateway and brokers. This is done by either the gateway or broker advertising themselves to the clients, or the clients active search for available brokers or gateways [80]. Once a connection is established between the client and the gateway, the clients can then send subscribing or publishing messages to the MQTT broker via the gateway [80]. In addition, the client maintains a list of active gateways [80].

To experiment with MQTT-SN on the Raspberry Pi, I used the Paho MQTT-SN Project developed by Tomoaki Yamaguchi and available on github. The repository contains an MQTT-SN Gateway and Client over XBee and UDP [100]. As an initial step, I built the Linux gateway on the Rasperry Pi with the commands shown in Listing 6.3.

By default, the gateway is built for UDP, and so, to create it for XBee, I had to set the [SENSORNET=XBEE] argument as illustrated in Listing 1 [100]. In addition, the gateway supports Mosquitto broker or any other MQTT broker. Once the gateway was built, MQTT-SNGateway, *.conf file and MQTT-SNLogmonitor files were copied into the ../ directory. The next step was to connect the gateway to some MQTT broker. To do this, I modified the gateway configuration file such that it connected to the hosted Mosquitto MQTT broker and listened to the configured serial port attached to the XBee radio device as shown in Listing 6.4. (Alternatively to using the hosted Mosquitto MQTT broker, I could have used a remote broker from cloud services such as the free public MQTT broker and CoAP server by Eclipse for testing available iot.eclipse.org.)

In order to test the full MQTT-SN architecture, I needed some MQTT-SN clients. The

```
BrokerName=localhost
BrokerPortNo=1883
BrokerSecurePortNo=8883

ClientAuthentication=NO
AggregatingGateway=NO
QoS-1=NO
Forwarder=NO
PredefinedTopic=NO
GatewayID=1
GatewayName=PahoGateway-01
KeepAlive=900

# XBee
Baudrate=9600
SerialDevice=/dev/ttyAMA0
ApiMode=2

# LOG
ShearedMemory=NO;
```

Listing 6.4: MQTT-SN gateway configuration file (Adapted from [100])

MQTT-SN library repository provides two types of MQTT-SN clients, one over XBee, running over Linux, Arduino and mbed (C and C++), and another one over UDP, running on Linux and Arduino. I chose to test the Arduino client over XBee. The client provides a library of files and an Arduino sketch (*soilmoisture.ino*) which can be used to test message transmission from the Arduino MQTT-SN client to the MQTT broker via the MQTT-SN gateway. Although the program offered a full solution of an Arduino MQTT-SN client running over ZigBee, the program was not packaged for use by the targeted user here. The program was complicated, it required a high C\C++ competence.

6.3 Conclusion

This chapter reported the last bit of experiments I conducted, with MQTT and MQTT-SN protocol. First I experimented with MQTT, and then MQTT-SN. At the end of the experiments it was concluded that the MQTT-SN protocol was not yet packaged for simplicity to beginners. The protocol implementations still required high competency in C\C++. As a result I decided to build a simplified ZigBee/MQTT gateway. The design of this gateway is discussed in the next chapter.

Chapter 7

yoGa Gateway: Design

As seen in the previous chapter, it was decided to build a simplified and light-weight ZigBee/MQTT gateway, able to receive sensor data from sensor nodes over ZigBee, and publish the data to an MQTT broker where other MQTT clients can consume the data. The gateway also receives response messages from the clients through the MQTT broker, and then forwards the data to the sensor nodes upon request. This chapter discusses the design of the bidirectional gateway which I named Yonella's Gateway ("yoGa").

7.1 Gateway Context

To start the design process, I drew the context in which the gateway will operate as shown in Figure 7.1. The drawing is based on the experiments I reported up to now in this thesis. As one can see, the gateway is going to be situated in the super-node.

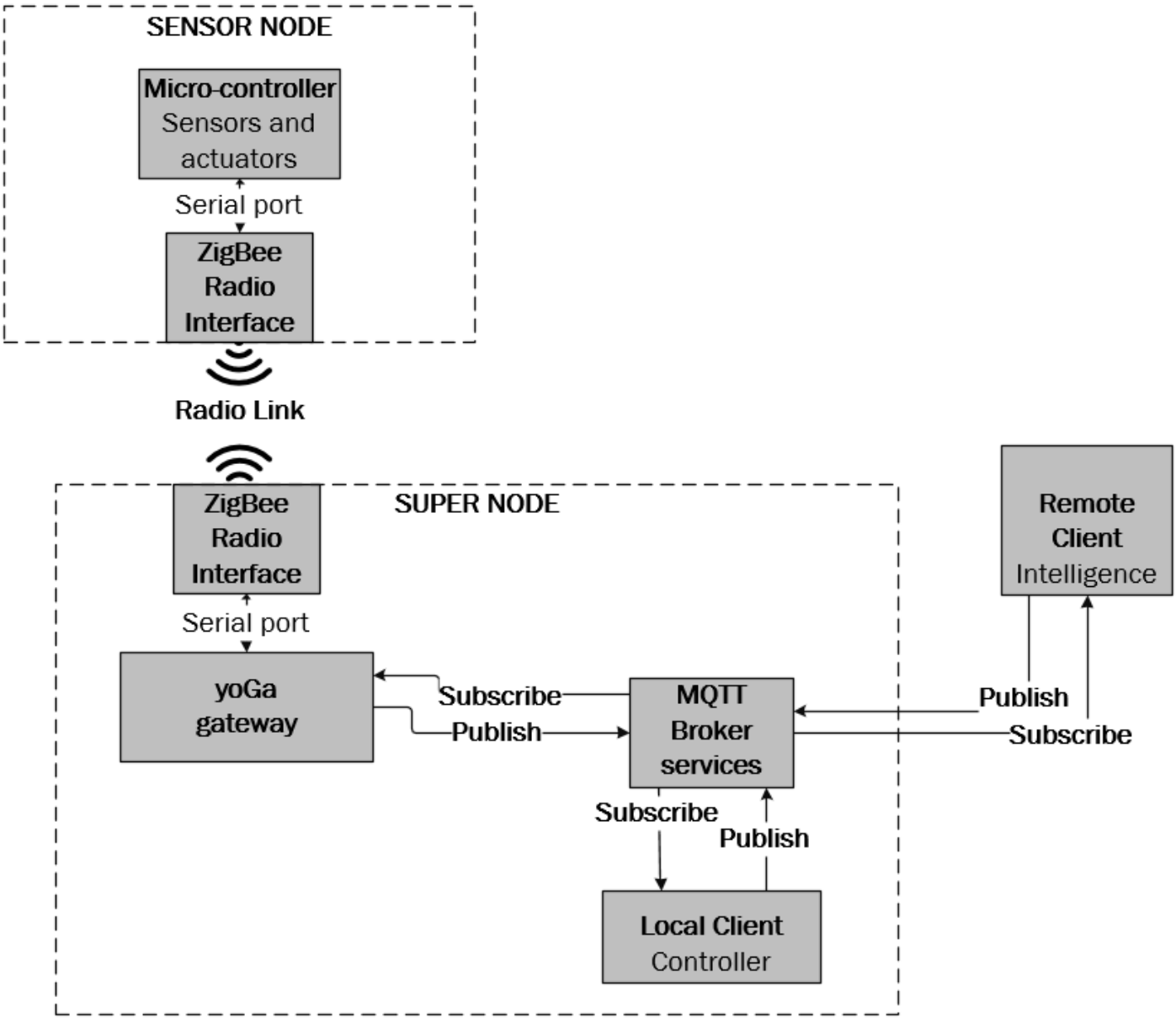

Figure 7.1: yoGa context diagram

7.2 High-Level Architecture

As illustrated by Figure 7.2, yoGa can be viewed as a ZigBee receiver and transmitter, coupled with an MQTT subscriber and publisher client. The ZigBee interface interacts with the sensor nodes in the ZigBee network. On the other hand, the MQTT interface interacts with standard MQTT protocol.

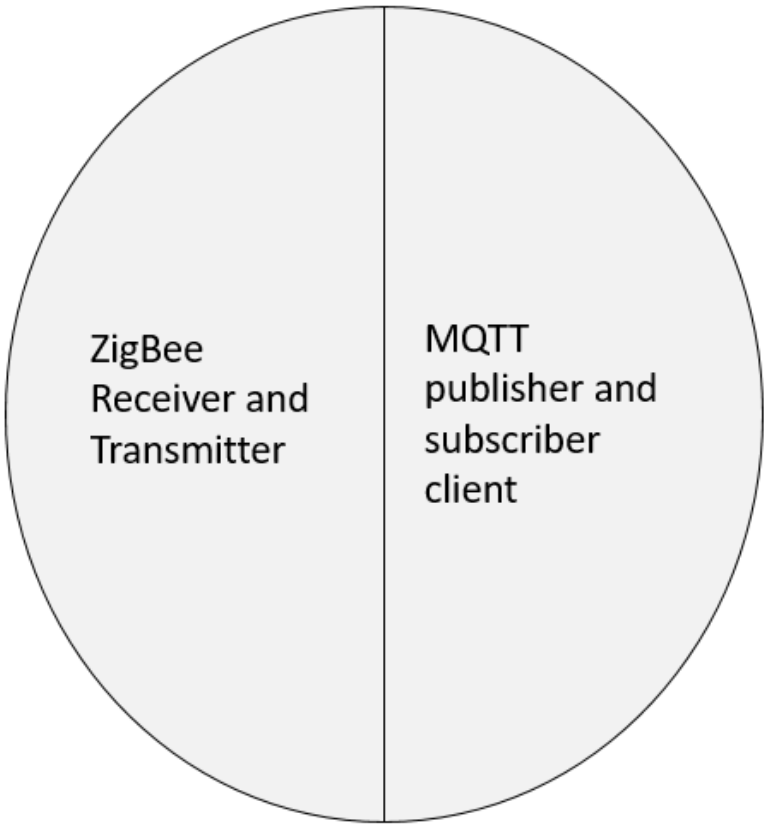

Figure 7.2: High-level architecture of yoGa

7.3 Design Constraints

The design of the gateway was constrained by the following. Firstly, I wanted the gateway to provide simplicity for the code in the sensor nodes, to make its use simple to beginners, as well as use computing devices at the node as simple as possible. Secondly, I wanted to guarantee the possibility of spending most time asleep for nodes that are not routers. Lastly, I wanted to implement the gateway as stateless as possible. The gateway is hosted by a Raspberry Pi microcomputer, which operates on limited memory (less than 1GB and an external SD card). Having the gateway implemented as a stateless protocol means that when it communicates with the nodes,it does not retain session state from previous requests. This then reduces memory requirements for the gateway. Having the gateway stateless is exceptionally useful for instances

where the gateway communicates with a larger network of nodes. In such a case, if the gateway was implemented as stateful, then it would have to retain the session state for every connection with every node, which is not a very wise option for a memory constrained environment. In addition to the memory, implementing the gateway as a stateless protocol makes easier the recovery from error condition that forces the gateway to restart. Unlike in a stateful protocol, where the server loses all its volatile state in an error condition, in a stateless protocol, the effects of server failure and recovery are almost unnoticeable, and makes easier the recovery from error condition that forces the gateway to restart.

7.4 Receiving and Handling a Message from a Sensor Node

If a sensor node (Node 1) sends a message with sensor data to some MQTT client through the ZigBee/MQTT gateway, and then expects a response after some time. One way of doing this would be for the node to send the message with the value of a sensor to the gateway, and then await a response containing a command for the associated actuation, for a set amount of time. Another would be for the node to send distinct messages, the first with the sensor data and the second requesting a command for the actuation. The latter method (as illustrated in Figure 7.3) mimics a publish/subscribe system and immediately fulfills two of the constraints mentioned- increased sleep time for the nodes and the gateway statelessness.

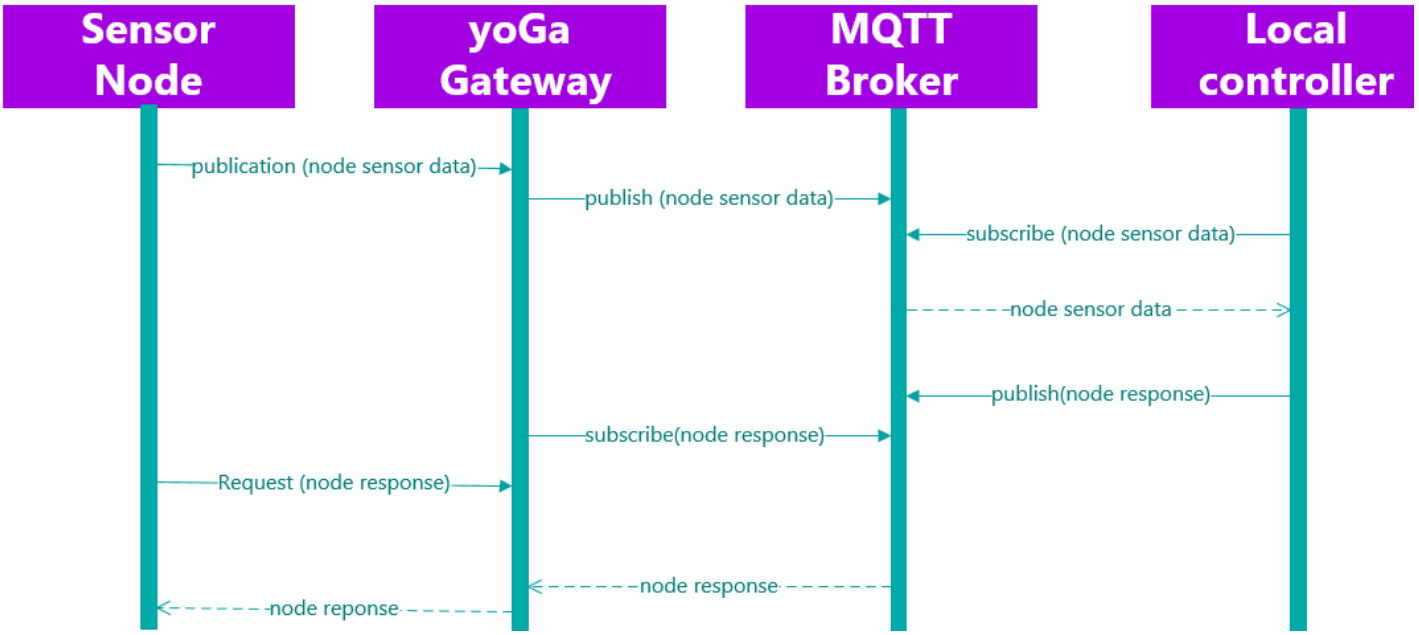

Figure 7.3: Data flows from a sensor node to a MQTT client through the yoGa gateway

7.5 ZigBee Acknowledgement Packets

When two ZigBee devices are communicating, the ZigBee protocol uses an acknowledgement mechanism. The protocol generates and sends an acknowledgement message to the sender device once the message is successfully transmitted to the receiving device. The ZigBee devices buffer messages in a small amount of memory, and the buffer works in a manner that the oldest message is on the front of the queue, and every incoming messages is added on the tail. This means that only the oldest message is visible for processing. Because the acknowledgement does not carry any application-relevant information, it is simply discarded. The pseudo-code of the algorithm running on the nodes is in Listing 7.1. Figure 7.4 is a finite state diagram describing a sensor node communication with yoGa, illustrating the different states of the node from sending a publication message to the gateway up to when the node receives a response from the gateway and taking action accordingly.

```
REPEAT Read sensor data
        Publish sensor data to gateway
        Wait for acknowledgement for 2 min
        if acknowledgement received
          discard acknowledgement from memory
        Send a request message to gateway
        Wait for a few seconds
        if received response from gateway
              decode response
              take action accordingly
              Wait for a few seconds
```

Listing 7.1: Pseudo code of a sensor node

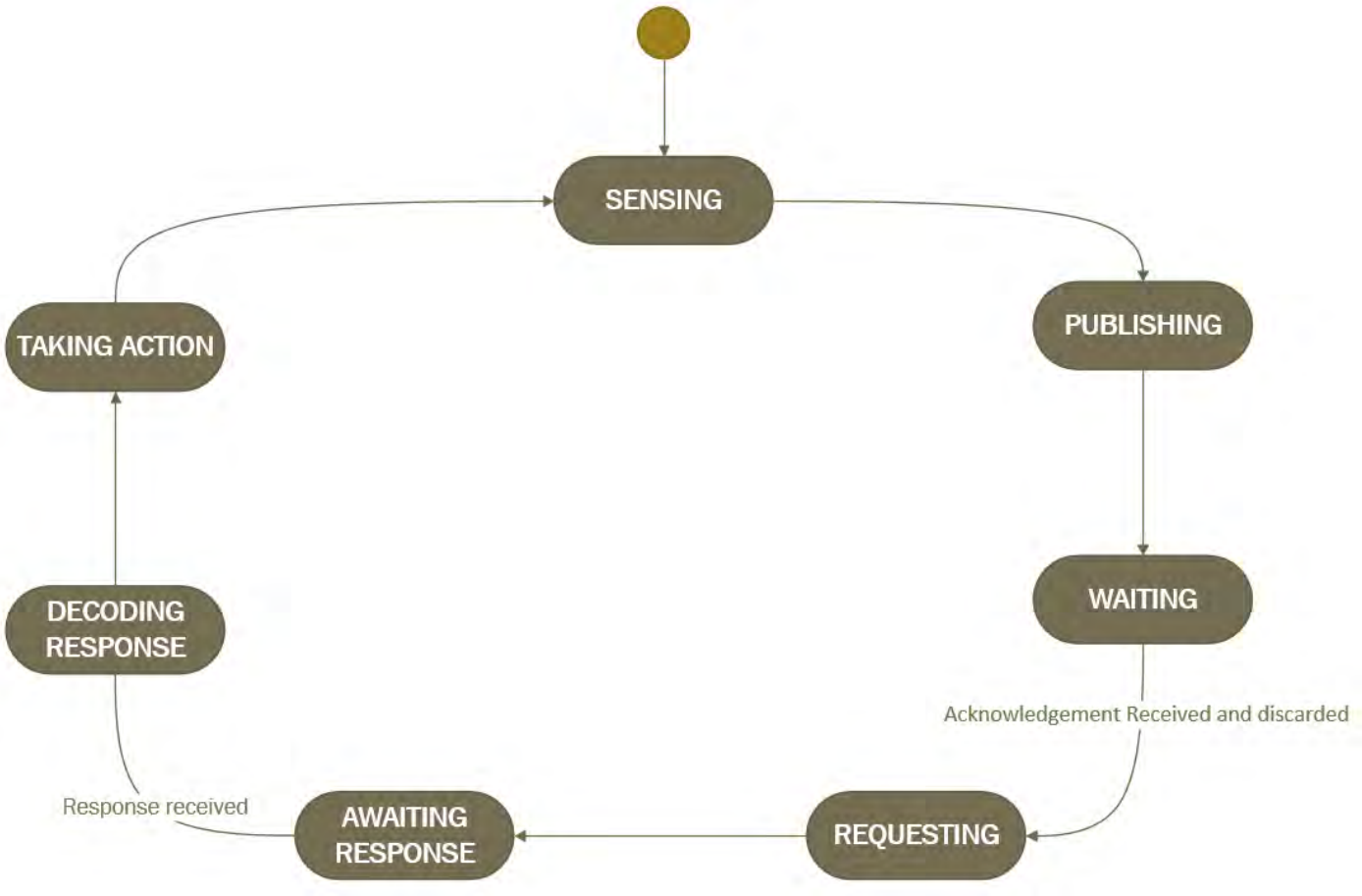

Figure 7.4: Sensor Node Finite-State Diagram

7.6 Gateway MQTT Topics

As illustrated in Figure 7.3, the gateway receives two types of messages from a sensor node, a publication and a request (which, in terms of MQTT, should be seen as a subscription). Let us now imagine that the gateway receives a publication message from a sensor node. In such a case, the gateway simply publishes the message to an MQTT broker on behalf of the sensor node, where the published data is available for consumption by any other MQTT client. MQTT brokers organize data in topics. As a result, upon receiving a publication message from a sensor node, the gateway creates a topic in the MQTT broker, and then publishes the received sensor data at the topic. For any other MQTT client to receive the published data, the client must subscribe to the topic in the broker, naturally.

Let us now imagine that the gateway receives sensor data from more than one sensor node. How is each sensor node data published in the MQTT broker such that it is easily identifiable by the other MQTT clients? In this case, as seen in Listing 7.2, the gateway has to create a unique topic for each node. To cater for this unique identification of each node topic, the gateway can utilize the ZigBee device MAC address, because in a ZigBee network, there are no nodes that can possibly share the same MAC address. And so, each time the gateway receives a publication message from a sensor node, the gateway prepares the node topic inclusing the node MAC address, and then publishes the node sensor data topic as it can be seen in the pseudo code in Listing 7.2 and in the finite-state diagram in Figure 7.5.

```
Create meta topic T_M
Subscribe to all nodes response topics
REPEAT Listening for messages
    if message m received
        if m = response from MQTT Client
            Save node response
        end if

        if m = publication  from node N
            Prepare node sensor data topic T_SN
                if N is not known by gateway
                Update T_M
                Publish node sensor data to MQTT server topic T_SN
             end if

                if N is known by gateway
                Publish node sensor data to MQTT server topic T_SN
            end if
        end if

        if m = request from node $N$
                if n is not known by gateway
                        Update T_M
                end if

                if n is known by gateway}
                    Retrieve node response
                        Send response to node N
                end if

        end if
```


Listing 7.2: Gateway pseudo code

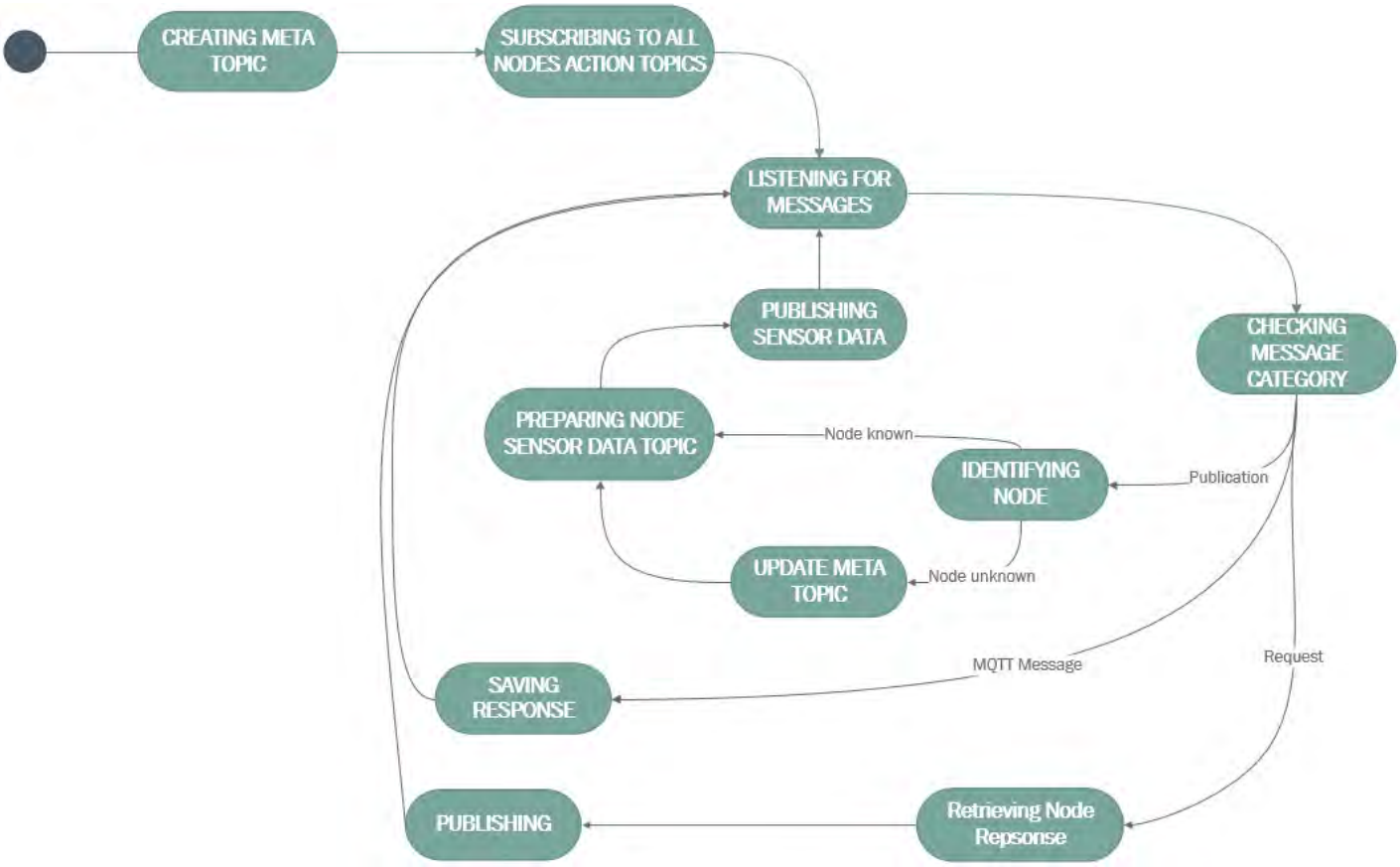

Figure 7.5: Gateway Finite-State Diagram

7.7 Receiving and Handling Data from MQTT Clients

Up to this point, I have explored how the gateway receives and handles data from sensor nodes, where the gateway simply publishes the sensor data to the appropriate node topic in an MQTT broker on behalf of the sensor node, and the data is consumable by other MQTT clients. Let us now imagine that some MQTT client would like to consume the published sensor data, and respond to the sender node through the gateway. For such an MQTT client to receive the sensor data, it would have to subscribe to the node sensor data topic created by the gateway, as discusses earlier. However, the sensor nodes are only visible to the gateway, the MQTT clients have no information about the nodes, e.g the total number of sensor nodes in the network, their MAC addresses, and other meaningful information about the nodes. As a result, it was decided to create a meta topic that will pass information about the sensor nodes from the gateway to

```
Subscribe to meta topic T_M created by gateway
Subscribe to all nodes sensor data topics (T_SN)
REPEAT Listening for MQTT messages
      if message received // caries node sensor data
        decode node sensor data
        prepare node response topic (T_AN)
        Run algorithm for node response
        publish node response to T_AN
```

Listing 7.3: Pseudo code of a local client

other MQTT clients. The gateway must therefore create this meta topic in the MQTT broker, and then publish the information about the nodes, e.g. the total number of sensor nodes in the system to the topic. In addition, the gateway updates the meta topic every time it receives a message from a new sensor node. When it comes to sending a response to a sensor node through the gateway, the same principle of subscribing and publishing applies. The client must then publish the response message in an appropriate topic in the MQTT broker as illustrated in Listing 7.3 Figure 7.6.

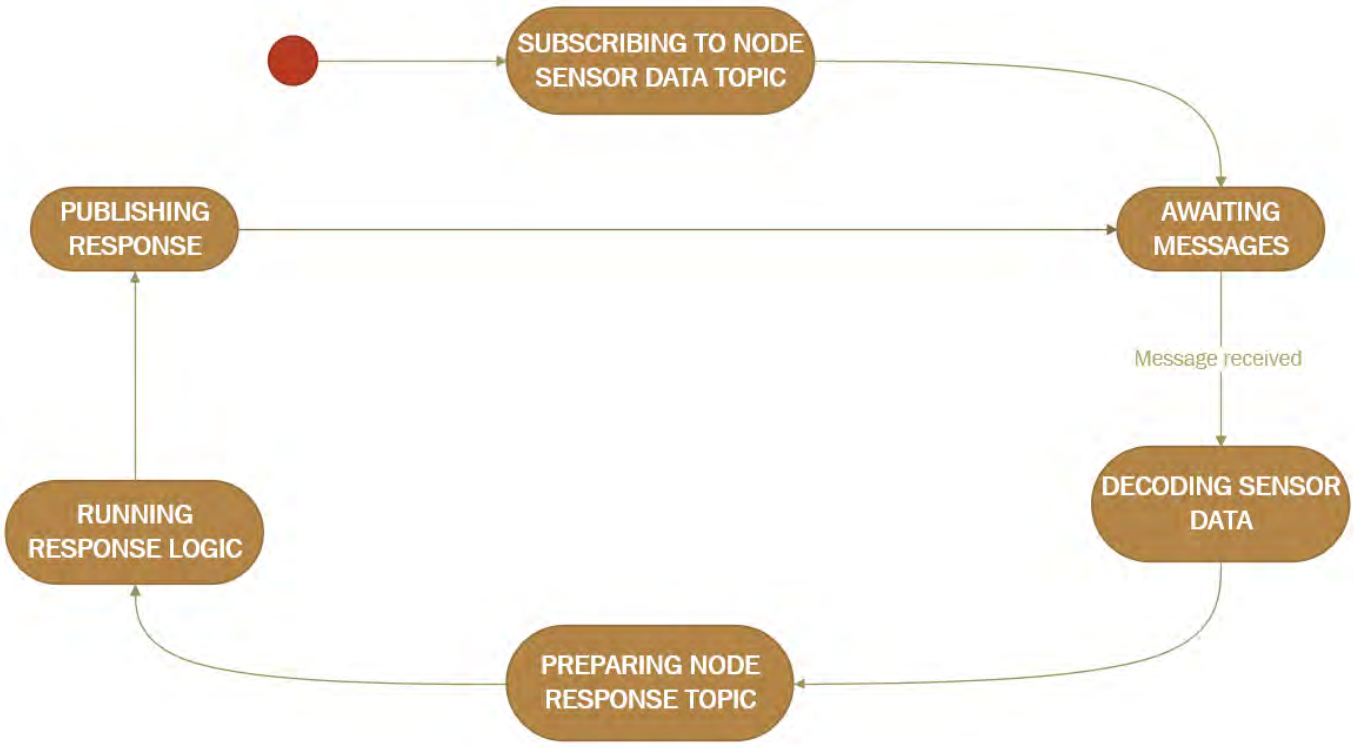

Figure 7.6: Local client Finite State Diagram

7.8 Transmitting an MQTT Client Response to a Sensor Node

Let us finally imagine that a client wants to send a response to a sensor node through the gateway. For this to happen, the gateway must have subscribed to the node response topic created by the MQTT client. So, the gateway receives publications from the MQTT client which needs to be delivered to a sensor node. However, as mentioned earlier, the gateway only sends a response to a node upon receiving a request message from that node (see Figure 7.3). As a result, the gateway should have a mechanism to keep the node response until the gateway receives a request message from the node, and only then transmit it. This is shown in Listing 7.2 and in Figure 7.5

7.9 Conclusion

This chapter reported on the design of the light-weight, simplified yoGa gateway that publishes and requests data to and from the MQTT clients on behalf of sensor nodes in a ZigBee network. The gateway was designed such that the ZigBee network component mimics the MQTT publisher/subscriber pattern, by sending separate messages to publish and to request data.

Chapter 8

yoGa Gateway: Implementation

In this chapter I discuss the implementation of the yoGa bidirectional ZigBee/MQTT gateway designed in the previous chapter, with accessory components to demonstrate the functionality of the gateway. These components are:

- sensor nodes,
- a local controller,
- and a remote database store

Some of the code in this implementation was either inspired by, or adapted from good code sources on the Internet, such as https://www.mongodb.com/blog/post/getting-started-with-python- and-mongodb. As a result, segments of the code presented in this chapter have strong similarities to ones found elsewhere. However, the code is used in a novel manner within the context of implementing a bidirectional ZigBee/MQTT gateway. Furthermore, the code is implemented with appropriate simplifications that are there to allow the testing and demonstration of the functioning of the gateway in a simple manner. The full implementation code will be examined in this chapter in segments. In appendix A are reproduced the various programs unsegmented for uninterrupted reading.

8.1 Tools and Components

I implemented the gateway and the accessory components using the hardware and software tools that I selected and experimented with in the previous chapters. Out of the tools, I selected the ones that I found easy to use. These hardware and software tools are listed in the following subsections.

8.1.1 Hardware

- Raspberry Pi 3
- Arduino Uno board
- Remote Computer
- 2 x XBee S2C Devices configured in a ZigBee mesh network

8.1.2 Software

- Arduino IDE
- Python IDE
- Arduino XBee library
- Python XBee library
- Python MQTT library
- MongoDB library
- Mosquitto broker

Figure 8.1 fills in the various components onto the drawing presented as 7.1 in the previous chapter.

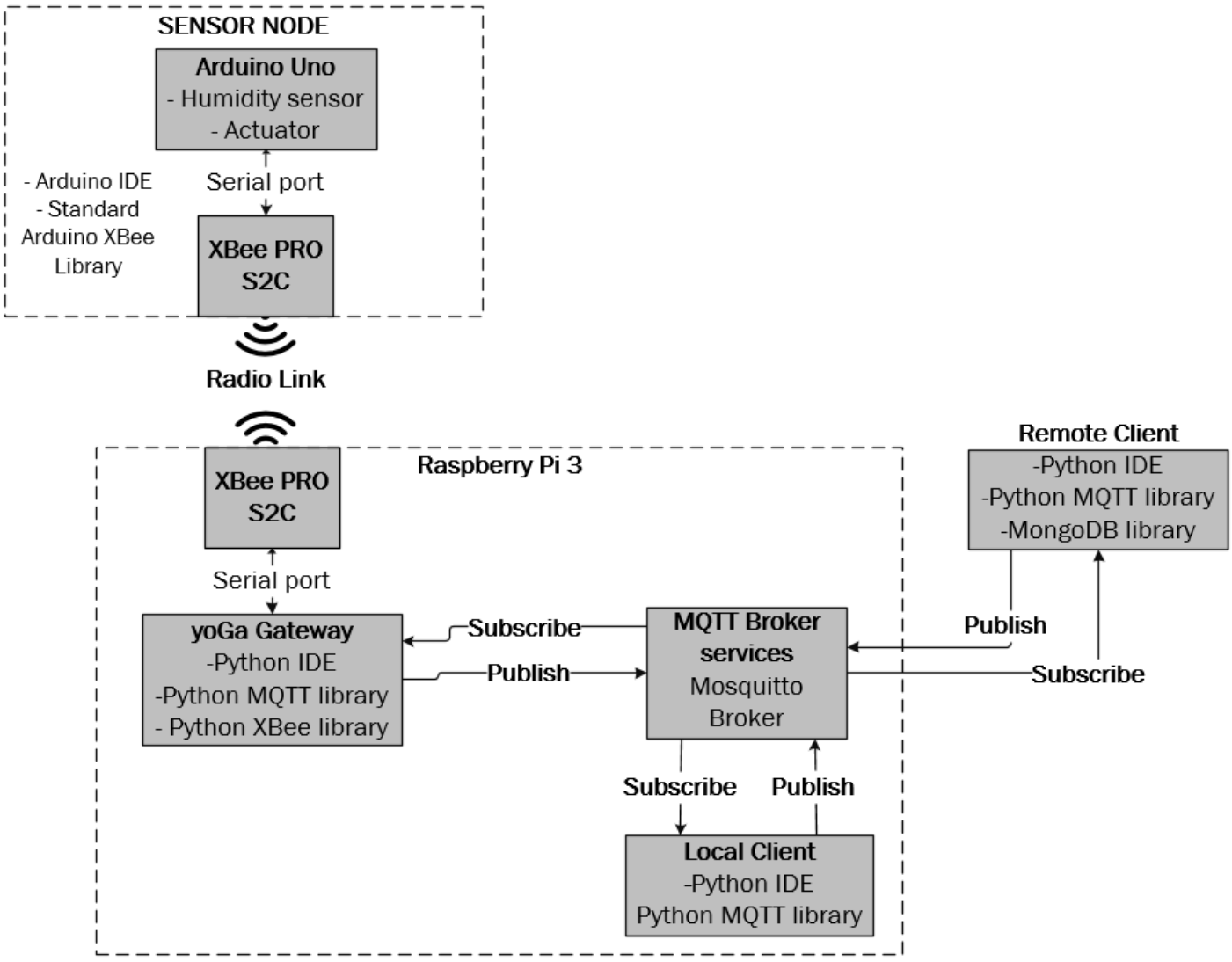

Figure 8.1: Illustration of the implementation tools and components

8.2 Overall Architecture

The gateway naturally consists of two interfaces, a ZigBee and an MQTT interface. The ZigBee interface listens to messages (publication or request) coming from the nodes in the ZigBee network. Through this interface, the gateway also sends to the nodes the actuation commands it receives from the MQTT clients via the MQTT interface. The gateway MQTT interface publishes the sensor data on behalf of the nodes to an MQTT broker. In this interface,

the gateway also listens to messages from the MQTT clients, which are then sent to the sensor nodes through the ZigBee interface as already discussed. The sensor nodes consist of a single ZigBee interface that sends a report message to the MQTT clients through the gateway, and then sends a request for action from the gateway. The local controller consists of an MQTT interface. In this interface, the controller listens for messages from the gateway, and publishes action commands to the gateway through the MQTT broker. The database saver consists of an MQTT interface, as well as an interface to a back-end database. The MQTT interface is a blocking call that listens to messages from other MQTT clients. The back-end interface simply saves the information carried by the MQTT messages into a running MongoDB database.

8.3 Data Structures

8.3.1 Nodes Representation in the Gateway

To represent the nodes in the gateway, I decided to use a dictionary. Dictionaries are generally good when you have a list of unique keys that map to values, where the entries are not in any specific order. In a ZigBee network, the nodes are identified by their unique MAC address. It was also important for the gateway and the rest of the system that the nodes are able to be uniquely identified. As a result, a dictionary structure was more suitable than an array or a plain list structure. The gateway uses this dictionary to temporarily keep incoming nodes data from the MQTT clients, until an action is requested by a node. The significance of this idea of only transmitting the data to the nodes upon request from the node is that it imitates a "subscription" in the MQTT protocol.

8.3.2 Nodes Representation in the Local Controller

To represent the nodes in the local controller, I decided again to use a dictionary for reasons that I have stated in the previous sub-section. However, in the controller, the dictionary keeps the agricultural information about the nodes, where each node is also identified by its MAC address. This node information includes the node GPS information, priority, valve status, type

of crop, and etc. The significance of this dictionary in this controller is that it keeps the state of the nodes. This way, the controller knows what nodes are currently being watered, and what nodes have the water switched off. For the scope of this implementation and testing, the dictionary is created in a simplified format and populated with example values. However, it can be recreated based on actual information from the field deployers and farmers, and then stored in a database.

8.3.3 Representing Nodes Values in the Database Saver

The database saver uses a dictionary that keeps incoming nodes humidity values. The significance of this dictionary is that the node humidity and action values do not arrive to the database saver at the same time. The node humidity value is published by the gateway to the MQTT broker, and is then consumed by the database saver and the local controller. The local controller then publishes the corresponding node action in the MQTT broker, and it is only then that the node action is consumed by the database saver (as well as the gateway). As a result, the dictionary of nodes is used to temporarily keep the node humidity value until the node action is received, and it is only then that the node value is stored in the database as a single entry.

8.4 yoGa

As I said, yoGa is a bidirectional MQTT/ZigBee gateway that receives messages from sensor nodes over the ZigBee protocol, and then publish the data carried by the messages in an MQTT server. The data is then consumed by other MQTT clients, which can also send back messages to the sensor nodes through the MQTT server via the gateway.

As illustrated in Listing 8.1, the gateway uses the following classes:

- An *XBeeDevice* to interact with the local XBee device physically attached to the Raspberry Pi,

```
from digi.xbee.devices import XBeeDevice  # local device, connected
    to Pi3
from digi.xbee.devices import RemoteXBeeDevice  # remote node,
    connect to Arduino
from digi.xbee.devices import XBee64BitAddress
import paho.mqtt.client as mqtt

import time  # sleep services
import binascii # to convert bytearray to hexadecimal string, using
    hexlify
```

Listing 8.1: Importing classes in the gateway

- a *RemoteXBee* device that is used to work with the remote XBee devices connected to the Arduinos in the sensor nodes,
- an *XBee64Address* that represents an address object of an XBee device, and will be used to represent the remote XBee devices,
- an *MQTT* client which contains functions connect, publish and subscribe to topics in a running MQTT server,
- *time* for sleeping services, and
- *banscii* which contains function to perform type conversion from a byte array to a hexadecimal string. This type conversion will be discussed in a greater detail later.

8.4.1 MQTT Subscription Handler

The gateway defined a callback function to handle the reception of MQTT messages originating from MQTT clients. The messages carry responses to be sent to sensor nodes by the gateway when requested by the node. As illustrated in Listing 8.2, the function takes three arguments, the *client* instance, the client *userdata*, and the actual message received, which is an instance

of the *MQTTMessage* class. The MQTT message consists of attributes such as the message topic, the message body, qos, etc. The function then extracts the address of the node to which the action is directed to, using the message topic string, and stores the message in a pre-initialized dictionary of nodes. MQTT topics follow a hierarchical structure, similar to folders and files in the file system, using the forward slash as a delimiter. The gateway relies on this hierarchical structure, using the node address and the data category to form the topic path. The node address identifies the sensor node to which the data belongs, and the data category assumes that one node can have more than one categories of data. This approach simplifies the process of subscribing to all nodes topics under the same category as it will be seen later. For simplicity, and maintaining the general settings of this research, the implementation of the gateway is in the context of smart irrigation. As a result, the topics in the gateway are in the form */humidity/node-Address* and */action/node-Address*, which indicate nodes humidity and action topics, respectively.

To extract the node address, the gateway first extracts the message topic in which the message was published, using the *message.topic* function, which returns the topic as a string. After extracting the node address, the function checks if the node is known by the gateway, that is if the node is in the dictionary of nodes. If the node is known, then the function loads the node response in the dictionary. The MQTT message payload is binary data and the dictionary stores the node response as a string. As a result, the function converts the response into a string, using the *decode* function, and then loads it in the dictionary for transmission to the node when requested. If the node does not exist in the nodes dictionary, then it means that the node is not in the ZigBee network as far as the gateway is concerned. Typically this situation might arise because that node has not yet transmitted after a restart of yoGa. The situation will normalize within an amount of time identical to a sensor node transmission cycle. In the worst case, still, it is an anomaly and so it is logged.

The gateway then initializes objects of the imported classes, starting with the *MQTT client* object using the *Client* function as seen in Listing 8.3. Following this, the gateway connects the client to the local MQTT broker, in my case running on the Raspberry Pi, using the *connect* function. The gateway then uses the hierarchical structure of topics to subscribe

```
def actionHandler(client, userdata, message):
    print("in handler")
    node = message.topic[8:24] #fetch the node address from the
message topic
    print("Received: action" + message.payload.decode("UTF") + "
from server for node" + node )
    if node in nodes: #check if the node is known by the gateway
                #if the node is known
      # put the action in dictionary for transmission when
requested by node
        nodes[node] = message.payload.decode("UTF")
    else:
        #error if the node is not known
      print("! error, unknown node !")
```

Listing 8.2: The function that handles the reception of MQTT messages in the gateway

```
client = mqtt.Client("Pi3-gateway3") #initialize an MQTT client
print("starting look")
client.connect("localhost") #connect client to local MQTT broker
client.loop_start() #to activate the callback functions
client.subscribe("/action/+") # subscribe client to all action
    topics
client.on_message = actionHandler #register the function to handle
    reception of MQTT messages
print("handler registered and active")
```

Listing 8.3: Initializing the MQTT client in yoGa, that subscribes to response topics in the MQTT server

the client to nodes response topics. Because the gateway is implemented in the context of smart irrigation, where a local controller sends node response messages through the gateway (this local controller will be explored in greater detail later). For testing purposes, the gateway subscribes the MQTT client to all nodes action topics. After subscribing the client to the topics, the defined handler function is linked with the client *on_message* callback so that the defined function handles each MQTT message received.

8.4.2 Listening to ZigBee Messages

Coming to the ZigBee portion of yoGa, the gateway initializes a local XBee object on the *AMA0* serial port of the Raspberry Pi, and opens the device connection as illustrated in Listing 8.4. The program logic execution is then suspended for 2 seconds to allow a safe completion of the connection (as seen in Listing 8.4).

The gateway then proceeds to its main loop, where it listens for ZigBee messages, and takes action accordingly. As stated in the Chapter 7, yoGa receives two types of message from a sensor node, a publication or a request for a response. A publication carries sensor data from the sensor nodes, which the gateway then publishes to an appropriate topic in an MQTT broker. On the other hand, a request message has to receive an answer. So, the gateway sends

```
#create a local XBee device object on serial port
localDevice = XBeeDevice("/dev/ttyAMA0", 9600)
#open device connection
localDevice.open()
# just allow the connection to complete, just in case
time.sleep(2)
```

Listing 8.4: Initializing an XBee object in the gateway

to the sensor node the action it received from an MQTT client, which in our case acts as the local controller.

With reference to Listing 8.5, the gateway uses the *read_data* function to read incoming messages in the local XBee device. The function call is blocking, meaning that it only returns to the program logic when the node receives a message from the gateway, or reaches time out. In my case, the gateway sets the timeout to 2 hours meaning that the it expects a message within 2 hours, which is correct in the demonstration context I have set up. Other context might require other timeouts. The function returns an *XBeeMessage* object, which amongst many attributes has the Remote XBee Device which contains information about the node that sent the message, and the message payload as a byte array.

The gateway then uses the remote device *get_64_bitaddr* function to extract the address of the remote XBee device that sent the message, as illustrated in Listing 8.5. The function returns the sender local device address as a property of *XBee64BitAdress* object. The *XBee64BitAdress* object contains the node address in a byte array which is not usable with the MQTT server. As a result, the gateway uses the *helixify* function in the banscii class, to convert the node address from a byte array into a hexadecimal string, which is a byte string, and is decoded into a string that is usable with the MQTT server using the decode function. Once the sender address is extracted as a string, the gateway checks if the node is known, that is, if it exists in the dictionary of nodes (see Listing 8.5). If the node is not known, then the node is simply added into the dictionary as discussed in the design chapter.

Following this, the gateway checks the type of message received which is either a publi-

```
while True:
  print("listening to zigbee messages")
#   block for 2 hours max;
#     (a message will arrive before time out)
  readMessage = localDevice.read_data(7200) #read a ZigBee message

# get the MAC address of the sender
  remoteAddress = readMessage.remote_device.get_64bit_addr()
# remoteAddress is a XBee64BitAddress object
# remoteAddress is a property of the class XBee64BitAddress,
# which contains the remote address in a bytearray;

# tranform remote address into string
  remoteAddressAsHexString = binascii.hexlify(remoteAddress.address)
# hexlify transforms it into a hexadecimal string - byte string

#  which must be decoded in order to be used with the MQTT server
  remoteAddressAsSString =  remoteAddressAsHexString.decode("UTF-8")

  # add the remote address to the nodes dictionary, if not there
  if remoteAddressAsSString not in nodes:
    nodes[remoteAddressAsSString] = "O"
```

Listing 8.5: Listening to ZigBee messages and identifying a ZigBee message sender

```
# the payload of the zigbee packet is a sequence of bytes;
#  if the first byte is 0 then it is a 'report' (publish) message
#  if it is 1, then it is a request for action (subscription)
    message

  #get the first byte of the payload byte array
  firstByte= readMessage.data[0]  # data is a bytearry

if (firstByte != 0 and firstByte != 1):
  #message is neither a report nor a subscription
      print("**************>>>>>>>>>>>>>>>>>>>> read to clear")
 # then discard it
```

Listing 8.6: Checking the type of message received from a ZigBee node

cation or an action request, or neither. To differentiate between a publication and an response request message, I set the first byte of the message payload from the sensor nodes to "0" and "1" respectively. This will be discussed further at a later stage. As a result, the gateway checks the first byte of the message payload data array. If the first byte is neither a zero nor 1, then the message is neither a publication nor a request, and is therefore read and discarded by the gateway, as seen in Listing 8.6.

If the first byte is zero, and so the message is a publication, the gateway will publish the sensor data that the message carries to the MQTT broker. As seen in Listing 8.7, to publish the sensor data, the gateway first prepares the data topic under which the message will be published in the form */humidity/node_address* as discussed earlier, using string concatenation. Following this, the node sensor data (which is humidity in this context) bytes are extracted from the XBee message payload byte array, using character indexing. The bytes are then converted into integer values which is the actual humidity value. The MQTT client is then connected into the local MQTT broker running on the Raspberry Pi, and then the node humidity value is published as a string in the prepared topic.

```
if firstByte == 0:  # it is a report, publish
  print ("received: humidity report from node " +
 remoteAddressAsSString)

  # prepare topic "/humidity/nodeMAC"
     topic = "/humidity/" + remoteAddressAsSString

  # get humidity value (0 to 1023)
  humidityBytes = readMessage.data[1:3]  # semantics of range!
  humidityValue = int.from_bytes(humidityBytes, byteorder = 'big')

  # pubblish!
  client.publish(topic, str(humidityValue)) #perform the publish
  print("humidity published to the server in topic"+ topic)
```

Listing 8.7: Handling a report message from a ZigBee node

On the other hand, if the first byte is a "1", and so the message is a request for action, the gateway retrieves the node response (which is an action on a valve that can be open or close in this context) from the dictionary of node actions. The action is then sent to the remote XBee node which is represented by the *remote_device*, loaded from the message topic earlier when the message was read (see Listing 8.8). However, if there is no node action in the dictionary, then again, the gateway throws an error message indicating that the node is not in the dictionary as as discussed earlier in section 8.4.1 (as seen in Listing 8.8). The main loop is repeated until the program terminates.

```
if firstByte == 1:
# it is a request for action, retrieve action from nodes
 dictionary
# and send it to the node

  #first check if the node is known by the gateway
  if remoteAddressAsSString in nodes:
    # then fetch the node actuation from the dictionary
    action = nodes[remoteAddressAsSString]
    print("received: action request from node " +
                              remoteAddressAsSString)
    # send; the remote node is represented by the remote_device,
    # which was loaded when the message was read
    try:
      #send the node action
      localDevice.send_data(readMessage.remote_device, action)
    except:
      print("Attention: not received!")
    else:
      print("sent: " + action + " to node "+remoteAddressAsSString
)
  else:
    print(" Error, node not in nodes dictionary !")
```

Listing 8.8: Handling a request message from a ZigBee node

8.5 Sensor Nodes

This section focuses on the code running on a sensor node in its interaction with the gateway. As previously discussed, the node sends a message carrying sensor data and a message request for an actuator action over ZigBee. To achieve this task, the sensor node uses the Arduino XBee library to talk to XBee radio devices. Within the XBee library the node uses objects of the following classes:

- *XBee*, which contains functions send and read incoming API packets
- *ZBTx* which represents a ZigBee request API packet
- *ZBRx* which represents a ZigBee response API packet
- *XBeeAddress64* which represents a 64-bit network address of an XBee device, and is set to hold the MAC address of the remote XBee device connected to the Raspberry Pi.

. Listing 8.9 illustrates the importing of these classes in the sensor nodes.

As illustrated in Listing 8.10, in its setup function, the sensor node initializes a serial port, which is then set as the serial port of the XBee object which was created earlier, using the *setSerial* function. In addition, the node sets the remote XBee device address of the request packet to the *XBeeAddress64* object which represents the XBee device in the Raspberry Pi, using the *setSerialAddress64* function. Finally, the node configures the digital pin 13 of the Arduino board as an output pin, which will represent an actuator. For simplicity and testing purposes, the node uses an LED as an actuator (switching it ON/OFF depending on the action received) instead of an actual water valve.

Before the loop function, three variables are created, one variable to hold a node soil moisture value, and two variables to hold the humidity MSB and LSB value, respectively. Again for simplicity, the node humidity value is not based on readings from an actual humidity sensor, but uses assigned values. The details of how the values are assigned will be discussed later.

The loop function is where the sensor node updates the humidity value, and sends messages to and handle incoming messages from the gateway over the ZigBee network. Listing

```
XBee xbee = XBee(); //creating an XBee object
//Remote XBee device address
XBeeAddress64 addr64 = XBeeAddress64(0x0013A200, 0x417DE0CB); //
    coordinator on super node ( pi 3)
//create ZBTX and ZBRX objects
ZBTxRequest zbTx = ZBTxRequest(); // this will be used to transmit
    outgoing packets
ZBRxResponse zbRx; // this will be used to read incoming packets
```

Listing 8.9: Sensor Nodes imported classes

```
void setup() {
  Serial.begin(9600);
  xbee.setSerial(Serial);
  //setting the zbTx remote device address to the coordinator
  zbTx.setAddress64(addr64);
  pinMode(13, OUTPUT);
  }
```

Listing 8.10: Sensor nodes setup function

```
void loop() {
//increment value by 1 each time is entered
sensorValue++;
//sensor value cannot exceed 1023
if (sensorValue > 1024) {sensorValue = 0;};
MSB = sensorValue / 256;
LSB = sensorValue % 256;
//populate the array that will be carried by the request packet
// 0 on the first byte means this is a publication
```

Listing 8.11: Populating sensor data

8.11 is an illustration of how the sensor node achieves this task and is a reference for the explanation in the rest of this section. The node uses stub logic to update the node humidity value by simply incrementing the humidity variable by 1 each time the loop is entered. Once the variable exceeds 1023, it is then reset to 0. The node uses these values between 0 and 1023 because a soil moisture sensor reads a 10-bit humidity value between 0 and 1023. The regularity of the humidity value increased was chosen to simplify the testing later.

The node sends two type of messages to the gateway, a publication and a request message, as discussed in the previous section. The node creates the data array to hold the payload which will be sent in the message as illustrated in Listing 8.12. The array consists of three elements. The first element of the data array indicates the type of message that is being transmitted, "0" to indicate a publication, and "1" to indicate a request for action. The second and third elements are the Most Significant Bytes (MSB), and the Least Significant Byte (LSB) of the humidity value. This data array is then set as the payload for the request packet which was created earlier, using the *setPayload* function. And then, finally, the *send* function is then used to send the report message to the gateway.

As shown in Listing 8.13, after sending the publication or report message, the node expects to receive ZigBee acknowledgement packet (which should be distinguished from a response from the gateway). The node uses the *readPacket* function to read and discard the acknowl-

```
//populate the array that will be carried by the request packet
 // 0 on the first byte means this is a publication
uint8_t data[] = {0,MSB,LSB};
 //set the array as payload of the request packet
 zbTx.setPayload(data);
 zbTx.setPayloadLength(sizeof(data));
   //sending the publication to the pi3
 xbee.send(zbTx);
```

Listing 8.12: Sending a report

edgement packet which carries no relevant information, with a specified 4 seconds timeout. After discarding the packet, the node delays the program for 2 seconds before proceeding to sending an action request message. These delays are only indicative. The general rule is to give enough time to the rest of the system to compute the actuator action in response to the publication of a humidity value. To send a request, the node sets the first byte of the data array into a "1", and then sends the request message.

Again, the node expects a ZigBee acknowledgement packet, which is also read and discarded. After sending the request message, the node waits for 2 seconds and listens for a response from the gateway within 4 seconds, using the *readPacket* function as shown in Listing 8.14. The node then decodes the packet payload which contains the node action ("0" or "1") and takes actuation accordingly. The node lights the LED to "open" the valve, and switches off the LED to "close" the valve.

```
  //Expects an acknowledgement packet after sending the publication
  xbee.readPacket(4000); // throw it away
  delay(1000);
  // wait 1 seconds before requesting  response of the controller
  //    (via sending a 'subscription' request)

// send subscritpion now
// 1 on the first byte means this is a subscription;
//proxy knows topic
  uint8_t ndata[] = {1,MSB,LSB};

  zbTx.setPayload(ndata);
  zbTx.setPayloadLength(sizeof(ndata));
  xbee.send(zbTx); //sending the subscription
  xbee.readPacket(4000); // throw the acknowledgement packet away
  delay(2000); //wait 2 seconds
```

Listing 8.13: Sending a Request

```
// and listen for the response from the gateway
 xbee.readPacket(4000);  // response should arrive within 4 seconds
//  load the packet into zbRx;
 xbee.getResponse().getZBRxResponse(zbRx);
//   now set LED depending on response from controller
 if (zbRx.getData(0) == 'O') {
          digitalWrite(13, HIGH);}
 else {
          digitalWrite(13, LOW);
          }
   delay(6000);  // wait 6 seconds before looping */
}
```

Listing 8.14: Handling a received from the gateway

8.6 Local Controller Client

The local controller is a local MQTT client that implements, in a simple minded manner, the local "decision point" in the system commanding the opening or closing of the water valve (other decision points can be located somewhere in the internet). The controller receives data from the sensor nodes through the MQTT broker from the ZigBee/MQTT gateway, and then replies with an action to control the node water flow. To achieve this task, the controller uses the following classes (see Listing 8.15):

- *mqtt.client*
- *mqtt.subscriber* which contains functions that allow a straight-forward subscribing and processing of messages, and will be used to subscribe the client to MQTT topics and register the callback function.
- *time*

The controller uses a dictionary that keeps the information of each node relevant for

```
import paho.mqtt.client as mqtt #import client
import paho.mqtt.subscribe as subscribe # for the callback
import time
```

Listing 8.15: Imported classes in the local controller client

taking the decision for an action, with each node identified by its MAC address as seen in Listing 8.16. This node information includes the node GPS information, priority, valve status, type of crop, and etc. For the scope of this implementation and testing, the dictionary is created in a simplified format and populated with example values. In reality, it will be created based on actual information from the field deployers and farmers, and then read in from a database.

To handle the arrival of all messages from the MQTT broker, a callback function was defined, as illustrated in Listing 8.17. In the function, the controller uses the *message.topic* method which returns the message topic as a string, and then extract the node address from the topic string using string indexing. Once the node address is extracted, the controller checks if the node is known, that is, if it exists in the dictionary of nodes. If the node exists in the dictionary, the node valve status value which indicates the node action ("O" for open, and "C" for close) is loaded from the dictionary, and the node action is published to the MQTT broker. To perform the publishing, the controller first connects the MQTT client object into the local MQTT broker using the *connect* function, and then prepares the node action topic in this format */action/node-address*. Finally, the action for node is published into the broker in the created topic, using the *publish* function. After publishing the node action, the controller runs a stub logic to update the node status in the nodes dictionary by simply changing it from "O" to "C", and vice versa. If the node does not exist in the dictionary of nodes, then the controller sends an error message indicating that the node is unknown by the controller.

As seen in Listing 8.18, the controller uses the *subscribe* callback function which offers a clean and straight-forward way of subscribing an MQTT client to a set of topics and process incoming messages using a user defined callback function. The function takes three arguments, the *on_message* callback function that handles all incoming messages, which in my case I set as the callback function defined earlier. The second argument that the function takes is the

```
nodes ={
  "0013a200417de439" :
  {
  "gps" : "",
  "crop" : "maize",
  "priority" : 3,
  "status" : "C",
  "etc" : ""
  },

  "0013a200417de0ba":
  {
  "gps" : "",
  "crop" : "maize",
  "priority" : 2,
  "status" : "C",
  "etc" : ""
    }
}
```

Listing 8.16: A dictionary that keeps agricultural information about sensor nodes

```
def reportHandler(client, userdata, message):
    node = message.topic[10:27] #fetch the node address from the
message topic
    if node in nodes: #checks if the node is known by the
controller
        print("working node action...")
        # run logic to determine what to do; here just a stub
        #load the node information from the dictionary of nodes
        info = nodes[node]
        action = info["status"] # and then fetch the node action
status valve "O" or "C"
        # publish the action to on the correct topic "/action/
nodeMAC"
        print("Received: node " + node + " humidity value " + action
)
        topic = "/action/" + node
        client.connect("localhost") # server running on this Pi
        client.publish(topic, action) #publish!
        print("published: node actuation to the local server")

        # change action, for easy testing of architecture
        if action == "O":
          action = "C"
        else:
          action = "O"
        # update the status in the dictionary with the info on the
processed node
        info["status"] = action
    else: # executes if the node is not in the dictionary of known
 nodes
        print("! error, unknown node !")
```

Listing 8.17: Callback function that handles MQTT messages in the controller

```
# let's register the callback, so that it receive all relevant
    messages
print("registering")
subscribe.callback(reportHandler, "/humidity/+", hostname="localhost
    ")
print("registered")
```

Listing 8.18: Subscribing an MQTT client to topics in the server

set of topics that the client subscribes to. My client subscribes to all node humidity topics in the MQTT broker. Lastly, the final argument is the MQTT host-name which represents the MQTT broker, I used the local running broker. The subscribe callback function call is blocking, which means that it listens and only returns to the flow of the program when the client receives a message from the MQTT broker. This eliminates the need to put a loop that listens to the MQTT broker until the client receives something. Although the function offers a simple and straight-forward way of subscribing a client to a set of topics, and handling incoming messages, it could not be used by the gateway program because I already had a blocking function call, the *readPacket* function.

8.7 Database Store Remote Client

The database store is a program that simply receives nodes humidity and action values from a sensor node and the local controller respectively through the MQTT broker, and then stores node values in a database. It is simply an example of how easily new logic can be added to the system. As illustrated by Listing 8.19, the client uses the following classes:

- *MongoDB*
- *mqtt.client*
- *mqtt.subscriber*

```
from pymongo import MongoClient

import paho.mqtt.client as mqtt
import paho.mqtt.subscribe as subscribe  # to be used the register
    the callback

import time  # sleep services
import binascii # to convert bytearray to hexadecimal string, using
    hexlify
```

Listing 8.19: Imported classes in the database store

- *time* and *banscii* for sleeping services, and data type conversions respectively, as discussed earlier.

The client uses a callback function to handle incoming MQTT messages, which is illustrated in Listing 8.20. In the function, I first check the category of the message, whether it carries humidity or action values. To obtain this information, the message topic was used. As stated earlier, topics are in the form */humidity/node-address* or */action/node-address*, and therefore incorporate information about the node address as demonstrated so far, and the category of data about the node. The client uses the *message.topic* function to extract the message topic string, and then uses string character indexing to extract the second character of the string, which is either an "h" or "a", indicating humidity and action values respectively. If the message category is humidity, then the node humidity value is stored temporarily in a dictionary of node humidities, which was created as a global variable. To store the humidity value, the client first extracts the node topic string, and then uses the *message.payload* function to get the node humidity value as binary data. The binary data is then converted into a string using the *decode* function, which is then finally saved as the node humidity value in the dictionary of humidities.

On the other hand, if the message category is action, that is it contains an action for a node, the node humidity and action values are stored in MongoDB. To store these values,

client prepares the collection string, which will be used later when performing an "insert". The collection string makes use of the node address, so, the client extracts the node address from the message topic string. Following this, the node loads the node humidity value from the dictionary of node humidities. The node action value is then loaded from the message payload as binary data, and then converted into a string using the *decode* function. Finally, the node humidity and action values are stored in the database (I will show how the database is created later).

As illustrated in Figure 8.21, the client then initializes an object of the MongoDB client, connected to a local running MongoDB. Following this, a database is created. This database is the where the actual node humidity and action values are stored as a collection as I had mentioned earlier. The MQTT client is then subscribed to all node humidity and action topics, and then registered the callback function using the subscribe callback function (see Listing 8.21)

```
# function to handle message from localController, received via the
    MQTT broker
def actionHandler(client, userdata, message):
        # debug
        print("in handler")
        print("%s : %s" % (message.topic, message.payload))
        print()
        category = message.topic[1]
        # debug
        print(category)

        if category == "h":  # humidity
          # save value to database it when an action arrives
          node = message.topic[10:26]  # topic in this case is /
    humidity/nodeMAC (16 characters long)
          print(node)
          humidities[node] = message.payload.decode("UTF")
          print(humidities[node])
        if category == "a":  # action
          # add humidity and action to database
          node = message.topic[8:24]
          print(node)
          humidityValue = humidities[node]
          valveValue = message.payload.decode("UTF")
          collection = "_" + node # this is to 'parametrize' the name
    of the collection
          db[collection].insert({"humidity": humidityValue, "valve" :
    valveValue})
```

Listing 8.20: Callback function to handle incoming MQTT messages

```
humidities = {} # create a dictionary that stores the node
    humidities
                                    #will be used to insert the node
    humidities and action to the database

# Connect to MongoDB
client = MongoClient(port=27017) # default port, on localhost
#    and create database
db=client.humidityAndValveValues2
# register actionHandler
subscribe.callback(actionHandler, ["/action/+" , "/humidity/+"],
    hostname="146.231.88.22")
```

Listing 8.21: Database store MQTT client subscribing to MQTT topics and registering function call

8.8 Testing

8.8.1 Unit and Integration Testing

The various components implemented in this chapter were tested. Firstly, each component was tested in isolation to verify that each component was working as desired. Following this, I tested their integration into the full system. Here, I tested the transportation of data from the sensor data to the MQTT broker through the gateway, and if the data was consumable by the MQTT clients. I also tested data transportation from the MQTT clients to the sensor nodes through the gateway. Indeed the data was successfully transmitted from the sensor nodes to the MQTT clients and vice versa without any problems. Testing was made easy by the decisions in the implementation, to use predicable values for independent variables (as for humidity values) or for the result of potentially complex computation (as for action values).

8.8.2 Decoupling of the Various Components

I tested the decoupling of the various components of the system. In this test, I switched off one node, and then switched it back on again. The gateway was happy to transmit and receive messages to and from the node without requiring any special re-entry procedure. I then switched off the local controller, and switched it back on. Again, the gateway was happy to transmit and receive messages to and from the controller. This is because the gateway is implemented as a stateless element and does it not retain any information or status about any client. Instead, the local controller is the one that keeps information of the nodes using the dictionary of nodes as I have explained earlier, and so is, by definition, the database store.

8.8.3 Intelligence of the System

It is important to note that in this chapter I only implemented and tested the functionality of the gateway, and how easily the sensor nodes can transmit data through the gateway without using any complex program running on them which was the case for the MQTT-SN protocol as discussed in Chapter 6.

As an aside, there is no implementation nor testing of the intelligence of the system as, discussed in Chapter 3, and according to the scope declaration in Chapter 1. Due to the decoupling of the components, one can easily add a client that will subscribe to the sensor nodes topics in the MQTT broker, and then use artificial intelligence, machine learning, and weather forecasting to improve the decision making of controlling the water flow in the sensor nodes. The client can then publish this information into the MQTT broker, where it can be easily consumed by the gateway.

8.9 Conclusion

This chapter reported on the implementation of the yoGa gateway using accessory components to illustrate functionality of the gateway in the context of a smart irrigation system in a very simple manner. This chapter implemented the smart irrigation system in a very simple manner, eliminated the more sophisticated components of the proposed system design in Chapter 3. yoGa implements a light-weight ZigBee/MQTT gateway that connects ZigBee sensor nodes into the standard MQTT in a simple manner. The gateway architecture enables the sensor nodes to send two different messages to publish and request data to and from the gateway, imitating the MQTT publisher/subscriber pattern. This architecture completely decouples the sensor nodes from the MQTT clients, which then allows all the various components of the system to act independently of each other. The gateway is implemented as a stateless protocol, allowing the various components to easily connect and re-connect without any special entry procedures. The gateway optimizes the an MQTT interface to the standard MQTT which allows the addition of components in the full system in a very simple manner. The components are simple MQTT clients that receive sensor data from the gateway through the MQTT broker, and are able to control the sensor nodes through the gateway. I initially thought that it would be a good idea to implement a meta topic that transmits meta information about the various nodes in the ZigBee network as stated in Chapter 8. However, this was later realized not to be fundamental in the gateway functionality. As a result, there was no implementation of such a topic in the gateway.

Chapter 9

Reflections on the Voyage

As stated in the Introduction chapter, the primary goal of my work was to investigate how easy or difficult it would be for a person with a Honours in Computer Science but no training in hardware, formal or informal, to use the needed components to build the field component of a smart irrigation system. In this chapter I present my experience and reflect on the ease or difficulty of each step towards the full system. I will first provide a brief description of who I am from an educational and technical point of view. Following this, I will examine each hardware and software component I used, together with the associated tooling, reporting on its maturity in terms of the support and documentation available, as well as its ease or difficulty of use.

9.1 My Profile

I am a Computer Science graduate with Honours from Rhodes University. I completed matric at Qhasana Secondary School, a school in a small village called Potsdam, which is just outside Mdantsane in the Eastern Cape, South Africa. My matric subject choices included Mathematics and Physical Sciences. After high school, I enrolled for a Bachelor of Science in Computer Science and Information System. In this program, I gained a good experience in general problem-solving with computer programming in C,C++ and C#, and a basic understanding of other programming languages such as Python and Java. After my undergraduate studies, I enrolled for an Honours degree in Computer Science, also from Rhodes University.

Through my Honors project, I gained experience in full stack web applications development, using front-end tools such as HTML, JavaScript, CSS, and back-end tools such as PHP. Prior starting this research work, I had no experience, formal or informal, with hardware and hardware interfacing. As a result, most of the components and tools that I used in this project were completely new to me. This led me to build solution in a bottom-up approach through a series of experiments, with each set of experiment advancing my understanding of the components and tools, and progressing to the solution. This was done after an initial top-down step, in which a draft of system design was put together and components and tools were selected. In the rest of the chapter, I will reflect on the maturity of each tool, as well as how easy or difficult the tool while building the solution.

9.2 Arduino

9.2.1 Maturity

I found the Arduino family of micro-controllers to be very well documented, mainly through the Arduino official page [7]. The page consists of a step-by-step guide for getting started with the different Arduino boards. The guide provides easy to follow instructions for setting up the Arduino, including links to download the Arduino software, installing drivers, getting the Arduino board to talk to the computer and simple Arduino examples. There is also community support available, which includes the Arduino Project Hub and Arduino Blog. These platforms allow programmers to share and view Arduino projects and examples for the different Arduino boards, from simple to difficult examples, including the project code and the circuit schematic. In addition to these platforms is the Arduino Forum, which offers project guidance and troubleshooting, and allows users to post questions.

I also found other great additional support and resources available online, which provide aid with the tools. These online platforms include the Makerspace [48] and the Instructables [39] websites, which are both parts of the larger Maker Movement as I had discussed in Chapter 2. These platforms provided simple projects, with an easy step-by-step guide, including code

snippets and pictures that equips beginners to learn through the projects.

The Arduino kit I used also comes with a copy of the Arduino Projects book, which I used as a guide to experiment with Arduino. Similar to the guide in the Arduino official page, the book provides a step-by-step guide for getting started with the Arduino as well as tutorials ranging from simple to more complex examples. The hardware tools are well explained in terms of what they are and how they work. Of course, this explanation was useful as an initial step to understanding what the tools were and what purpose they are used for. Each tutorial in the projects book includes a very well detailed guide on how to build the electric circuit. The guide clearly highlighted what tools I would need to build the electric circuit for the tutorial, as well as a detailed and easy-to-follow guide of how to place the components in the electric circuit board. Each tutorial also contains a well-explained Arduino code to control the hardware elements, where each line of code is clearly explained.

9.2.2 Ease of Use

As a beginner to physical computing and micro-controller, building electronic circuits boards was a completely foreign concept for me. I conducted a series of experiments from the tutorials in the Arduino Projects book as I had stated earlier. Initially, I had no knowledge of what the various hardware tools were, or even how to assemble them into an electric circuit board. As a result, I initially relied on the guide provided by the projects book to direct me on how to assemble the different components and build an electric circuit. My ability to build the electric circuits improved the more I did the tutorials to a point that I was able to build them myself, and confirm with the guide whether I built the circuit correctly. The process of learning to build these electic circuit boards did not take me long. In a couple of days I had gained an understanding of how they work, with the more exercises I did.

The Arduino software is based on a simplified version of the C\C++ programming language. This simplified version hides complex micro-controller processes. For example, in one button click ("upload" button), Arduino code is compiled and uploaded into the Arduino

board. All the events and processes involved during the compilation and uploading are running in the background and are hidden from the programmer as I had already stated in Chapter 4. Arduino standard programs are called sketches and are based on a predefined structure. They are made up of at least two parts, the setup and the loop functions. This predefined structure provides a guideline to make the programming as simple and straight forward as possible. The Arduino software contains libraries that add functionality to Arduino sketches. These libraries are easily added into the Arduino sketch, simply by using the "include" statement at the top of the Arduino program. I made use of the Arduino XBee library to talk to our XBee modules through the Arduino serial port. Overall, given my programming background, learning the Arduino language was a fairly simple task, it was a matter of adjusting to the new semantics. However, although the language is based on a simplified version of the C\C++, the learning experience may be completely opposite experience for someone with no programming background, compared to mine. They might require more time to learn and understand the language.

9.3 Raspberry Pi

9.3.1 Maturity

As already mentioned in Chapter 2, Raspberry Pi is a series of small and low-cost computers that was developed to teach programming concepts [92] [54]. Raspberry Pi has gained recognition and success, especially in the Maker Movement. Raspberry Pi boards contain a 40-pin or 26-pin header general purpose input/output (GPIO) depending on the model, which can be classified in software as input or output pins, and are usable for a wide range of purposes, including connecting the Raspberry Pi to electric circuits and sensors. Raspberry Pi can run various operating systems, including the official Raspbian OS, Ubuntu and Debian. Raspberry Pi boards consists of USB ports for connecting keyboards and mouses, and some boards consists of a full size HDMI and 4 USB ports that can be used to connect a TV or monitor using a HDMI cable so that the micro-computer operates as desktop. Raspberry Pi consists of various generations of boards from the first generation to the forth (to date), where each generation

improves from the previous one making a more sophisticated micro-computer. Raspberry Pi is well supported in its official page [68].

9.3.2 Ease of Use

Raspberry Pi is a family of micro-computers. Raspberry Pis are able to function as a computer if one does not need great computing power. The Raspberry Pi board itself is not open hardware, but it is based on the open-source ecosystem. The Raspberry Pi runs a variety of Linux distributions such as Raspbian and Ubuntu, the latter being the most supported and also open-source. The Raspberry Pi came empty, with no operating system installed, and as a result, I had to set it up before I could start using it. To set up the Raspberry Pi, I first installed an operating system, a completely new task to me, mainly because computers nowadays come with the operating system pre-installed. However I was able to set up the OS in the Raspberry Pi in a few minutes with the great support that was available online including the Raspberry Pi official page [68], with step-by-step instructions on how to install the operating system onto the Raspberry Pi. I also had to configure the serial port through which the Raspberry Pi will use to send and receive messages from the XBee devices. I completed this task within a few minutes, using a few commands that I found from online support.

The Raspbian operating system is best controlled via the command line based, as opposed to the GUI-based Windows OS which I was accustomed to. Switching from using the Windows operating system to Raspbian meant adapting to working with the command line. I found great support online to support this transition, including the Raspberry Pi official page [68], and over some time, I got accustomed to this different way of working.

On the Raspberry Pi, I used the Python language to write the smart irrigation system program, including the yoGa gateway. The language is supported by the Raspbian OS which I had running on my Raspberry Pi. This time, given my programming background, learning the language was a simple task. However,the experience may differ for someone with no background in programming. The language official page [67] offers support for programming beginners

including a beginner's guide, interactive courses, and tutorials for younger beginners. I also found other resources for getting started with the language. These include the w3schools website [88]. The website offers a learning guide for the language. It teaches the language concepts with exercises and tutorials in between to assess understanding. These resources can be use by first time programmers to learn the language.

9.4 XBee ZigBee Devices

9.4.1 Maturity

XBee ZigBee devices implement the ZigBee protocol, including the ZigBee mesh. ZigBee is a very well-designed, reliable, and mature for use. The protocol supports multiple network topologies: point-to-point, point-to-multipoint, star networks, and most importantly mesh networks. ZigBee mesh networks are self-forming and automatically heals itself if nodes are removed or disabled. The protocol is well documented on online resources such as the ZigBee Alliance and the Digi official pages.The devices are well documented on the official Digi page[22] through a Digi product kit user guide. The guide includes step-by-step instructions for setting up and configuring XBee devices. The Digi page also includes support through the Digi Forum, where users post or respond to questions. I also found support for setting up our XBee devices available on online platforms such as the Instructables [39] and the Sparkfun [79] website. The platforms provided tutorials, with a step-by-step guide for configuring our XBee devices. XBee devices are supported by libraries which allow programs to communicate with the devices in API mode. I found a few available XBee libraries supporting various languages, including Python and Arduino. These libraries allow programs to communicate with XBee devices configured in API mode. On the Arduino, I used the xbee-arduino library, and the Python XBee library on the Raspberry Pi. The XBee library was well documented and supported on the official Digi page [22]. I also found the library to be very well packaged and mature for use as it contains functions to interact with XBee devices in a simple manner. I found the Python XBee library to also be well-documented and well-supported on the official XBee Python Library page [23], and the library was also very well-designed and mature for use.

9.4.2 Ease of Use

Configuring XBee Devices

I configured the XBee devices using the XCTU software on a windows desktop. Configuring the modules in this fashion required no programming at all, instead, I just set a few parameters on the software interface. ZigBee networks are self-formed once the roles of the various nodes are defined, so, I had to configure a coordinator node through which the other nodes will join the network, setting the network PAN ID parameter. Adding the other nodes in the network was also simple: I just set the network PAN ID parameter to the same value I set for the coordinator node and the role of the node in the network (router or end device). In my opinion, configuring these devices required no computer science background, one can easily configure the devices by simply following the tutorials available online.

I used our Arduino boards as a USB-serial device, to interface between the XBee devices and the computer. To configure the modules in such a manner, I had to make sure that the communication between the computer and our XBee devices completely bypasses the Arduino bootloader. To achieve this, I had to upload an empty Arduino sketch into the Arduino board, each time I wanted to configure a node. Also, I had to set the Arduino XBee shield switch to the "USB" position, so that the computer could communicate directly with the XBee module. My decision to use the Arduino was mainly because I already had them at hand. I later realized that I could have simply used an actual USB-to-serial board instead, which would have been less complicated. I would simply connect the one side of the adapter the computer, and the other side to the XBee device, and have the computer communicating directly with the XBee module.

XBee Transparent and API Mode

XBee ZigBee devices offer a simple serial interface that acts as a serial line replacement, this is known as the transparent mode. However this interface has limitations, it only works for point-to-point and point-to-multipoint communications, where you have one node communicating to

only one node, or many nodes communicating to one central node. Of course this communication was not going to suit my ultimate objective of the nodes forming a mesh network to help route messages. I then discovered the powerful ZigBee API which is well designed making available the full power of the ZigBee network to the application programmer, base on a well designed ZigBee mesh network protocol. The API mode organizes messages in packets, and is rather more complex than the AT mode.

XBee Libraries

As mentioned earlier that I utilized the Arduino and Python XBee libraries. The libraries contain easy and straight-forward functions to interact with the XBee devices configured in the advanced API mode.

9.5 MQTT Protocol

MQTT is a well-designed protocol, and it is fairly mature for use. The protocol implements good and simple decoupling through the publisher/subscriber messaging pattern. MQTT is very well documented in the official MQTT official page [55]. As an open standard, there are many open-source implementations of MQTT servers in different programming languages, some of which are documented on the official MQTT page. There is also a great number of open-source MQTT clients available for various programming languages including Python and Arduino, as well as command line clients also documented on the official MQTT page.

9.5.1 Mosquitto MQTT project

Maturity

For the purpose of my implementation, I used the Mosquitto MQTT project, an open-source message broker implementing the MQTT protocol versions 5.0, 3.1.1 and 3.1. The project also

implements command line, as well as a C and C++ libraries of MQTT clients to subscribe and publish messages. Mosquitto is highly portable and available across a variety platforms such as Windows, Mac, Ubuntu, Debian and Raspberry Pi. The project is well documented in its official page [26]. I found useful support available for the project, for example there is a Github repository where users can report bugs and share their modifications. The repository also contains a simple guide to get started with installing the broker as well as examples of publishing messages and subscribing to messages in the broker. There is also a mailing list where users can talk to each other and share ideas.

Ease of Use

The Mosquitto project is a good and simple implementation of the MQTT protocol, I was able to set up and run the server in a short space of time. I also experimented with the command line MQTT clients, in which publishing and subscribing to topics in the server was also very simple and straight forward. To publish data, at minimum, I needed to know the broker IP address and the message. To subscribe to a topic, I needed to know the broker IP address, and the topic I wanted to subscribe to. Setting up and implementing the project MQTT command line clients was a very easy task. In a few minutes of experimenting with the command line clients, I was able to fully understand the protocol publishing and subscribing mechanisms using the support available online. I believe that someone with no computer science background can also set up and test the project command line MQTT clients using online tutorials and documentations.

9.5.2 MQTT Libraries

Maturity

As I mentioned earlier, that there are many open-source implementations of MQTT client libraries available for different programming languages. During my experiments and implementation of the ZigBee/MQTT gateway, I used the Eclipse Paho MQTT Python client library on the Raspberry Pi. The project is an open-source standard that implements versions 5.0, 3.1.1, and 3.1 of the MQTT protocol on Python 2.7.9+ or 3.5+. The library is also portable across

different platforms such as MacOS, Windows, and is also available on the Python Package Index as well as the GitHub repository. and is well-documented on the Eclipse Paho official page [27].

Ease of Use

The Paho MQTT Python client library provides a client class that allows applications to connect to an MQTT broker, and then subscribe to topics, or publish messages. The library uses simple and very straight-forward functions to initialize the class object, publish messages, and subscribe to topics. Installing the library and experimenting with the clients was a very easy and fast task. Also, my programming background was an advantage, and a result, for someone with no background in programming, the eperience may differ.

9.6 MQTT-SN Protocol and Implementations

To experiment with MQTT on the Arduino, I used the PubSub client available on the Arduino IDE, which implements an MQTT client that is able to send and receive messages an MQTT broker. However, I have stated in previous chapters that MQTT generally requires a sophisticated transport layer protocol such as the TCP/IP stack. As a result, I could not use the client on my Arduino programs since there I transmit and receive data using the ZigBee implementation in XBee PRO S2C. This meant that I had to look for a different client such as the one in MQTT-SN. MQTT-SN is a superset of the MQTT protocol, which was specifically designed to support wireless and non-TCP/IP networks such as UDP and ZigBee. MQTT-SN introduces a new type of device, the gateway, which is responsible for protocol conversion between the MQTT-SN clients and the MQTT broker. As a result, in addition to the MQTT clients, I looked for an implementation of an MQTT-SN gateway on the Raspberry Pi, over ZigBee.

9.6.1 Maturity

I found a few MQTT-SN projects available on the GitHub repository, which implemented MQTT-SN gateway and clients. One of these project was developed by Ian Craig under the Eclipse Paho project. The project implemented a Linux gateway over XBee, UDP6 and LoRa link, as well as MQTT-SN C\C++ clients on embedded platforms. However, I discovered the clients were incomplete and as a result, I could not use the project to experiment with MQTT-SN. The second project I discovered implemented an MQTT-SN gateway and clients for specific Arduino models and Linux. Although the project supported a wide range of transmission technologies such as Ethernet, WiFi, ZigBee, LoRa, and BLE, the clients did not support my specific Arduino model, Arduino Uno and as a result, I could not use this project to experiment MQTT-SN on Arduino. The last project I found was one written by Tomoaki Yamaguci, also under the Paho project. The project implemented an MQTT-SN gateway over XBee or UDP, running on Linux, an MQTT-SN client over XBee, running on Arduino, Linux and mbed, as well as an MQTT client over UDP, running on Linux and Arduino. This project almost met my requirements from a maturity point of view, except that the project has been inactive since its last maintenance in 2017. The project was well documented in the GitHub repository, with a step by step guide for setting up the gateway as well as the client library. There was also support available on the GitHub community to ask questions, report errors or bugs, and even share changes or improvements.

9.6.2 Ease of Use

With regards to code for the third MQTT-SN project, provided by Tomoaki Yamaguci, getting the MQTT-SN gateway running on the Raspberry Pi and connected to the running Mosquitto broker was doable task through the assistance of the guide. To set up the gateway, I had to make a few changes in the configuration file so that I set it up for my specific requirements. For example, I had to change the broker name to the local running Mosquitto broker, broker port number to the default "1883", serial device the specific one I was using, "dev/ttyAMA0", and lastly, the XBee device API mode to "2". Coming to the Arduino MQTT-SN client library,

the library consisted of a full solution of an Arduino MQTT-SN client running over ZigBee. However, unlike the other libraries that I had until then, the library was not mature enough for the level of easy use that I was looking for. The library was unrefined, in order to understand its workings it required high competency in the C++ language. As I said, the library has not been maintained since 2017, and it is not maturing to this day. As a result, I decided not to use the library and build my own simplified yoGa gateway, that is able to act as a bridge between the ZigBee and MQTT clients.

9.7 Handling Hardware

As a beginner to working with hardware, I encountered a few instances where I mishandled the hardware. What follows are two little examples. While building the circuits, I often confused the LEDs anode and cathode legs, and as a result, the program would not behave as expected. There was also a point where I powered the Raspberry Pi from an old micro-USB cable that I had connected to a desktop and using the built-in micro-USB port. After a short while, the Raspberry Pi was starting erratic behavior, i.e freezing now and then, and later the Raspberry Pi would not boot at all. I then learned the this was a result of under-powering the Raspberry Pi not giving it enough current. Raspberry Pi boards have specific power supply requirements, my specific boards (model 3) requires 2.5A currency, and clearly, the USB cable was supplying less power than needed. As I result, I reloaded an image of the operating system into the SD card, and repeated the process of installing the operating system into the Raspberry Pi. Naturally, I changed the power supplier to a Raspberry Pi micro USB power supplier, to avoid running into the same problem again.

There was also a point where one of the XBee devices had their antenna broken. I then learnt that did not completely destroy the XBee device, it simply decreases the device transmission range. I then decided to take advantage of the broken antenna, and use the device to test the ZigBee mesh in a small environment, where I had the device communicating to another XBee device that is out of its range through an intermediate XBee device. Of

course, instead of chopping off the XBee device antenna, I could easily reduce the XBee device transmission power exactly at configuration.

9.8 Conclusion

In this chapter, I evaluated the maturity of the hardware and software tools I had selected to build the field of a smart irrigation system in terms of the support and documentation available for the tool. I also reflected on my overall experience, ease or difficulty of working with each one of the tools as a computer science graduate from Rhodes University with a strong programming background, and no formal/informal training in hardware computing.

Chapter 10

Conclusion

10.1 Thesis Summary

In this report, I first discussed the research background and context, the scope of the project, the methodology that the research follows and the structure of the report. Following this, I explored the larger 4IR and two of its technological underpinnings that can be used to build a smart irrigation system, namely, the Internet of Things and Artificial Intelligence. I also reviewed existing similar work, first smart irrigation projects using IoT components, and secondly projects investigating the adoption of IoT systems in developing countries and the general ease or difficulty of building IoT systems for first time users. This review broadened my understanding of the problem at hand. I then proceeded to analyse the problem in a classical top-down fashion guided by a series of scenarios. The analysis enabled me to break down the problem in order to identify the abstract components that I would need to build the solution. The solution was designed as a series of sensor nodes, placed across a field, with the ability to measure the moisture of the soil and transmit the moisture value to a central "super node". The super node then controls each sensor node water flow, transmitting the appropriate actuation command back to the node. I wanted the system nodes to be able to communicate with a super node as easily and reliably as possible. As a result, I was looking for a wireless communication solution that offers mesh capabilities, as well as automatic route discovery and advanced self-healing capabilities. I also wanted for a data transfer protocol that decouples the sensor nodes from the

super node, to eliminate dependencies between the nodes and allow them to act independently of each other, i.e. send messages regardless of whether or not the other node is active. So, the ZigBee communication protocol and the MQTT data transfer protocol were selected. Following this, I selected and explored the off-the-shelf hardware and software components which are suitable to implement the identified abstract components of the solution. I decided to use the open-source Arduino family of micro-controllers, and a Raspberry Pi micro-computer. I also chose the commercially available MH-Series soil moisture sensor, and an electrovalve as an actuating component to control the water flow. I decided to use the commercially available family of XBee RF modules which implement the ZigBee protocol, as well as the open-source Mosquitto MQTT project that implements an MQTT broker and a C library of MQTT clients. I then experimented with each one of these tools, where I first learnt the tool while developing the solution. During the experiments, I also examined the maturity of each tool in terms of documentation and support available for the tool. I found the a majority of the tools beginner-friendly, packaged well with plenty of support available online. I then found out that the MQTT protocol required a sophisticated transport layer protocol such as TCP in the IP stack, and so MQTT was not usable with sensor nodes supporting only the ZigBee protocol. As a result I decided to use a super set of the MQTT protocol, MQTT-SN, which was designed specifically to support ZigBee and UDP, while naturally maintaining the MQTT pub/sub model. I discovered an MQTT-SN library available on GitHub. The library implements Arduino MQTT-SN client over the ZigBee protocol, and an MQTT-SN Linux gateway. I then experimented with this library, and it was concluded that the MQTT-SN was not packaged nor matured enough for the targeted user. MQTT-SN required a high level of understanding of C\C++, and had not been maintained since 2017, and it is not maturing to this day. As a result, I decided to build my own simplified ZigBee/MQTT gateway, called yoGa. The gateway was implemented and shown to easily takes sensor data from a sensor node, and publish it to an MQTT broker, where the data is consumable by other MQTT clients. The gateway was implemented with three other accessory components to build the field components of a smart irrigation system, as well as to illustrate its functionality.

10.2 Goals Revisited

The primary goal of the research work reported here was to investigate how easy or difficult it would be to build the field component of a smart irrigation system for a computer science graduate without any formal or informal hardware experience. This goal was broken down into, and achieved through the realization of three sub-goals: The first sub-goal was to run a scan of the domain in order to understand what has been done to create such a solution, and to identify components that can be used to build the solution. During this exploration I found plenty of projects done to build smart irrigation system using IoT components. The second sub goal was to select the off shelf and inexpensive components that can be used to build the solution. This sub-goal was achieved through the realization of the previous sub-goal, as well as the top-down process through which the problem was broken down to identify the abstract components needed to build a smart irrigation system as discussed in Chapter 3. Once the abstract components were identified, the off-the-shelf hardware and software components suitable to implement these components were selected. The third sub-goal was to investigate how easy it would be for a general computer science graduate with no formal or informal hardware experience, to assemble the selected tools a smart irrigation system. The sub-goal was actually realized by building the smart irrigation system using the selected components through a series of experiments as stated in the previous section. During the experiments, I constantly examined the ease and difficulty of each components, as well as the maturity of each component in terms of documentation and support available. I found most of the selected tools to be indeed very user friendly and packaged well for a person of my education background, with the exception of the MQTT-SN protocol as discussed in earlier. Because of that, I built a gateway, ZigBee/MQTT, which I named Yoga.

The outcome of the investigation is that such an IoT project is within the reach of a computer science graduate who has followed the standard computer science curriculum at Rhodes University. It must be added, however, that it would have helped quite a bit having a course in computer hardware and embedded systems.

10.3 Future Work Recommendations

At the end of the research, I had managed to build a working prototype of the yoGa gateway, and implemented it in the context of a smart irrigation system as stated already. For future work, I recommend for the gateway to be transformed from a prototype into a fully robust software of industrial strength that can be easily used by anyone in any context.

The implementation of the smart irrigation system focused on transporting information between the sensor nodes and MQTT clients and vice versa, which was done successfully. I did not explore on using actual soil moisture values or an actual irrigation logic, but used stub logic in order to test the data communication paths. So, a natural next step would be to investigate the back end data processing in order to build a sophisticated logic module which optimizes water usage and crop growth. The system can also utilize connection to the bigger Internet through the Raspberry Pi to access weather forecasting (e.g precipitation) to provide an improved decision support for the irrigation, so that the system takes decisions based on both the soil moisture and the plants' water needs, and also considers the weather forecasting in the area. This functionality realizes a complete IoT architecture, adding a cloud computing component. Cloud computing can apply data analytics such as visualization schemes to show device performance, identify inefficiencies, and come up with ways to improve the system. It also offers capabilities to draw patterns and correlations on historical data which can further contribute to creating algorithms to improve the system automation, through machine learning.

Bibliography

[1] James E Addicott. "The Precision Farming Revolution". In: *The Precision Farming Revolution*. Springer, 2020, pp. 1–35.

[2] Adepu, Ashok and Saideep, Ch and Krishna, Y. and Srishaka, P. "ZigBee Based Self-Organizing Network". In: *International Journal of Engineering Trends and Technology* 34.4 (2016), pp. 194–198.

[3] Oracle South Africa. *What is the Internet of Things?* Online. [Last Accessed: Jan 2019]. URL: https://www.oracle.com/za/internet-of-things/what-is-iot.html.

[4] Mohammad AlMarzouq et al. "Open Source: Concepts, Benefits, and Challenges". In: *Communications of the Association for Information Systems* 16.1 (2005), pp. 756–784.

[5] Rapp Andrew. *XBee-Arduino*. Online. [Last Accessed: September 2019]. 2016. URL: https://github.com/andrewrapp/xbee-arduino.

[6] P Aravind et al. "A Wireless Multi-Sensor System for Soil Moisture Measurement". In: *2015 IEEE SENSORS*. IEEE. 2015, pp. 1–4.

[7] Arduino, SA. *Arduino*. Online. [Last Accessed: February 2019]. URL: https://www.arduino.cc/.

[8] Arduino.cc. *Built-In Examples: Read Analog Voltage*. Online. [Last Accessed: August 2020]. URL: https://www.arduino.cc/en/Tutorial/BuiltInExamples/ReadAnalogVoltage/.

[9] Arduino.cc. *Getting Started:Introduction*. Online. [Last Accessed: February 2019]. 2019. URL: https://www.arduino.cc/en/Guide/Introduction.

[10] Desmond Tutu Ayentimi and John Burgess. "Is the Fourth Industrial Revolution Relevant to sub-Sahara Africa?" In: *Technology Analysis & Strategic Management* 31.6 (2019), pp. 641–652.

[11] RASS Bannatyne and Greg Viot. "Introduction to Microcontrollers". In: *WESCON/97 Conference Proceedings*. IEEE. 1997, pp. 564–574.

[12] Bernadette Johnson. *How the Raspberry Pi Works*. Online. [Last Accessed: October 2019]. 2019. URL: https://computer.howstuffworks.com/raspberry-pi2.htm.

[13] Vongsagon Boonsawat et al. "XBee Wireless Sensor Networks for Temperature Monitoring". In: *Second Conference on Application Research and Development (ECTI-CARD 2010), Chon Buri, Thailand*. 2010, pp. 221–226.

[14] Kevin Carillo and Chitu Okoli. "The Open Source Movement: a Revolution in Software Development". In: *Journal of Computer Information Systems* 49.2 (2008), pp. 1–9.

[15] T.A. Chowdary et al. "Effective Implementation of Low-Cost Smart Irrigation System". In: *International Journal of Innovative Technology and Exploring Engineering* 8.6 (2019), pp. 1805–1810.

[16] Codebender Authors. *How to Use XBee Modules As Transmitter Receiver - Arduino Tutorial*. Online. [Last Accessed: July Nov 2018]. URL: https://www.instructables.com/How-to-Use-XBee-Modules-As-Transmitter-Receiver-Ar/.

[17] Burak H Çorak et al. "Comparative Analysis of IoT Communication Protocols". In: *2018 International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE. 2018, pp. 1–6.

[18] Bernstein Corinne. *MQTT (MQ Telemetry Transport)*. Online. [Last Accessed: June 2020]. URL: https://internetofthingsagenda.techtarget.com/definition/MQTT-MQ-Telemetry-Transport.

[19] Alessandro D'Ausilio. "Arduino: A Low-Cost Multipurpose Lab Equipment". In: *Behavior Research Methods* 44.2 (2012), pp. 305–313.

[20] S Darshna et al. "Smart Irrigation System". In: *IOSR Journal of Electronics and Communication Engineering (IOSR-JECE)* 10.3 (2015), pp. 32–36.

[21] DD Dasig. “Implementing IoT and Wireless Sensor Networks for Precision Agriculture”. In: *Internet of Things and Analytics for Agriculture, Volume 2*. Springer, 2020, pp. 23–44.

[22] Digi International Inc. *IoT Solutions, Software, Products, Services*. Online. [Last Accessed: February 2019]. URL: https://www.digi.com/.

[23] Digi International Inc. *XBee Python Library*. Online. [Last Accessed: February 2019]. URL: https://xbplib.readthedocs.io/en/latest/.

[24] *Digi XBee ZigBee*. Online. [Last Accessed: February 2019]. URL: https://www.digi.com/products/embedded-systems/digi-xbee/rf-modules/2-4-ghz-modules/xbee-zigbee.

[25] Dale Dougherty. “The Maker Movement”. In: *Innovations: Technology, governance, globalization* 7.3 (2012), pp. 11–14.

[26] Eclipse Foundation. *Eclipse Mosquitto*. Online. [Last Accessed: February 2019]. URL: https://mosquitto.org/.

[27] Eclipse Foundation. *Eclipse Paho - MQTT and MQTT-SN software*. Online. [Last Accessed: February 2019]. URL: https://www.eclipse.org/paho/clients/python/docs/.

[28] Robert Faludi. *Building wireless sensor networks: with ZigBee, XBee, arduino, and processing*. O’Reilly Media, Inc., 2010.

[29] Daniel Fisher and Gould Peter J. “Open-Source Hardware Is a Low-Cost Alternative for Scientific Instrumentation and Research”. In: *Modern Instrumentation* 1.2 (2012), pp. 8–20.

[30] Ala Al-Fuqaha et al. “Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications”. In: *IEEE Communications Surveys & Tutorials* 17.4 (2015), pp. 2347–2376.

[31] Carlos Gonzalez. *What's the Difference Between Pneumatic, Hydraulic, and Electrical Actuators?* Online. [Last Accessed: May 2019]. 2015. URL: https://www.machinedesign.com/mechanical-motion-systems/linear-motion/article/21832047/whats-the-difference-between-pneumatic-hydraulic-and-electrical-actuators.

[32] Jayavardhana Gubbi et al. "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions". In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660.

[33] Jorge Guerra Guerra and Armando Fermin Perez. "Alignment of Undergraduate Curriculum for Learning IoT in a Computer Science Faculty". In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 2017, pp. 362–362.

[34] Erica Rosenfeld Halverson and Kimberly Sheridan. "The Maker Movement in Education". In: *Harvard Educational Review* 84.4 (2014), pp. 495–504.

[35] M Hillman. "An Overview of ZigBee Networks A guide for implementers and security testers". In: *MWR InfoSecurity* (2016), pp. 1–12.

[36] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. "MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks". In: *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*. IEEE. 2008, pp. 791–798.

[37] Dogan Ibrahim. *PIC32 Microcontrollers and the Digilent chipKIT: Introductory to Advanced Projects*. Oxford: Newnes, 2015.

[38] Ahmed Imteaj et al. "IoT Based Autonomous Percipient Irrigation System using Raspberry Pi". In: *2016 19th International Conference on Computer and Information Technology (ICCIT)*. IEEE. 2016, pp. 563–568.

[39] *Instructables: Yours for the making*. Online. [Last Accessed: February 2019]. URL: https://www.instructables.com.

[40] Suprabha Jadhav and Shailesh Hambarde. "Android Based Automated Irrigation System using Raspberry Pi". In: *International Journal of Science and Research* 5.6 (2016), pp. 2345–51.

[41] Sagar Jaikar and R. Kamatchi. "A Survey of Messaging Protocols for IoT Systems". In: *International Journal of Advanced in Management, Technology and Engineering Sciences* 8.2 (2018), pp. 510–514.

[42] Tariq Jamil. "RISC versus CISC". In: *IEEE Potentials* 14.3 (1995), pp. 13–16.

[43] Foughali Karim, Karim Fathallah, and Ali Frihida. "Monitoring System Using Web of Things in Precision Agriculture". In: *Procedia Computer Science* 110 (2017), pp. 402–409.

[44] Bruce Kogut and Anca Metiu. "Open-Source Software Development and Distributed Innovation". In: *Oxford review of economic policy* 17.2 (2001), pp. 248–264.

[45] Roger T Koide et al. "Plant Water Status, Hydraulic Resistance and Capacitance". In: *Plant Physiological Ecology*. Springer, 1989, pp. 161–183.

[46] Guoping Li, Yun Hou, and Aizhi Wu. "Fourth Industrial Revolution: Technological Drivers, Impacts and Coping Methods". In: *Chinese Geographical Science* 27.4 (2017), pp. 626–637.

[47] Roger A Light. "Mosquitto: server and client implementation of the MQTT protocol". In: *Journal of Open Source Software* 2.13 (2017), p. 265.

[48] *Makerspaces*. Online. [Last Accessed: February 2019]. URL: `https://www.makerspaces.com/`.

[49] Mirjana Maksimović et al. "Raspberry Pi as Internet of Things Hardware: Performances and Constraints". In: *Design Issues* 3.8 (2014).

[50] Paul Malmsten. *Python-XBee Documentation*. Online. [Last Accessed: April 2020]. 2013. URL: `https://python-xbee.readthedocs.io/en/stable/genindex.html`.

[51] Vachirapol Mayalarp et al. "Wireless Mesh Networking with XBee". In: *2nd ECTI-Conference on Application Research and Development (ECTI-CARD 2010), Pattaya, Chonburi, Thailand*. 2010, pp. 10–12.

[52] James O Mergard et al. *Flexible Microcontroller Architecture*. US Patent 6,415,348. 2002. URL: https://patents.google.com/patent/US6415348B1/en?q=Flexible+Microcontroller+Architecture&oq=Flexible+Microcontroller+Architecture.

[53] *Meshlium -The IoT Gateway-Waspmote Technical Guide*. Online. [Last Accessed: January 2021]. URL: https://development.libelium.com/waspmote-technical-guide/meshlium.

[54] Iqbal Mohomed and Prabal Dutta. "The Age of DIY and Dawn of the Maker Movement". In: *GetMobile: Mobile Computing and Communications* 18.4 (2015), pp. 41–43.

[55] MQTT Organisation. *MQTT - The Standard for IoT Messaging*. Online. [Last Accessed: February 2019]. URL: http://mqtt.org/.

[56] Victoria Namirimu. "User Requirements for Internet of Things (IoT) Applications: An Observational study". MA thesis. Sweden: Blekinge Institute of Technology, 2015.

[57] Susana Nascimento and Alexandre Pólvora. "Maker Cultures and the Prospects for Technological Action". In: *Science and Engineering Ethics* 24.3 (2018), pp. 927–946.

[58] Sharmila Nath, Jayanta kumar Nath, and Kanak Chandra Sarma. "IoT Based System for Continuous Measurement and Monitoring of Temperature, Soil Moisture and Relative Humidity". In: *Technology* 9.3 (2018), pp. 106–113.

[59] Sofia Amador Nelke and Michael Winokur. "Introducing IoT Subjects to an Existing Curriculum. An Ongoing Experience at the Faculty of the Technology Management - HIT". In: *Cyber Physical Systems. Model-Based Design*. Springer, 2018, pp. 193–196.

[60] Ndung'u Njuguna and Signé Landry. "The Fourth Industrial Revolution and Digitization will Transform Africa into a Global Powerhouse". In: *Foresight Africa: Top Priorities for the Continent 2020 to 2030*. Brookings Institution, 2020, pp. 61–66.

[61] Olugbenga Kayode Ogidan, Abiodun Emmanuel Onile, and Oluwabukola Grace Adegboro. "Smart Irrigation System: A Water Management Procedure". In: *Agricultural Sciences* 10.01 (2019), pp. 25–31.

[62] Alex Oliszewski, Daniel Fine, and Daniel Roth. *Digital Media, Projection Design, and Technology for Theatre*. Routledge, 2018.

[63] Sethi Pallavi and Smruti R Sarangi. "Internet of Things: Architectures, Protocols and Applications". In: *Journal of Electrical and Computer Engineering* 2017.1 (2017), pp. 1–25.

[64] Sofia Papavlasopoulou, Michail N Giannakos, and Letizia Jaccheri. "Empirical Studies on the Maker Movement, a Promising Approach to Learning: A literature Review". In: *Entertainment Computing* 18 (2017), pp. 57–78.

[65] Kylie Peppler and Sophia Bender. "Maker Movement Spreads Innovation One Project at a Time". In: *Phi Delta Kappan* 95.3 (2013), pp. 22–27.

[66] Alison Powell. "Democratizing Production Through Open Source Knowledge: from Open Software to Open Hardware". In: *Media, Culture & Society* 34.6 (2012), pp. 691–708.

[67] Python Software Foundation. *Welcome to Python*. Online. [Last Accessed: February 2019]. URL: `https://www.python.org/`.

[68] Raspberry Pi Foundation. *Raspberry Pi- Products*. Online. [Last Accessed: February 2019]. 2015. URL: `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`.

[69] Ammar Rayes and Samer Salam. "The Things in IoT: Sensors and Actuators". In: *Internet of Things From Hype to Reality*. Springer, 2017, pp. 57–77.

[70] Rishabh, Jain. *How to Interface XBee Module with Raspberry Pi*. Online. [Last Accessed: July 2019]. 2019. URL: `https://circuitdigest.com/microcontroller-projects/raspberry-pi-xbee-module-interfacing`.

[71] Chandan Kumar Sahu and Pramitee Behera. "A Low Cost Smart Irrigation Control System". In: *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*. IEEE. 2015, pp. 1146–1152.

[72] Klaus Schwab. *The Fourth Industrial Revolution*. USA: Crown Publishing Group, 2017.

[73] Klaus Schwab. *The Fourth Industrial Revolution: What it Means, How to Respond*. Online. [Last Accessed: May 2020]. URL: `https://www.weforum.org/agenda/2016/01/the-fourth-industrial-revolution-what-it-means-and-how-to-respond/`.

[74] Michael Shiloh Scott Fitzgerald. *Arduino Projects Book*. Italy: Arduino LLC, 2012.

[75] Levent Seyfi, Ertan Akman, and Tuğrul C Topak. "Smart Irrigation System". In: *World Academy of Science, Engineering and Technology, International Journal of Electrical and Computer Engineering* 2 (2015).

[76] Warren Snyder and Monte Mar. "Programmable Microcontroller Architecture (Mixed Analog/Digital)". US Patent 6,724,220. 2004. URL: https://patents.google.com/patent/US7221187.

[77] *Soil Moisture Sensor – How to choose and use with Arduino*. Online. [Last accessed: March 2020]. 2019. URL: https://www.seeedstudio.com/blog/2020/01/10/what-is-soil-moisture-sensor-and-simple-arduino-tutorial-to-get-started/.

[78] Nisha Ashok Somani and Yask Patel. "Zigbee: A Low Power Wireless Technology for Industrial Applications". In: *International Journal of Control Theory and Computer Modelling (IJCTCM)* 2.3 (2012), pp. 27–33.

[79] *Sparkfun Electronics*. Online. [Last Accessed: February 2019]. URL: https://www.sparkfun.com/.

[80] Andy Stanford-Clark and Hong Linh Truong. "MQTT For Sensor Networks (MQTT-SN) Protocol Specification Version 1.2". In: *International business machines (IBM) Corporation* 1.2 (2013), pp. 1–27.

[81] Cope Steve. *Simple Python MQTT Publish and Subscribe Example Script*. Online. [Last Accessed: June 2020]. 2020. URL: http://www.steves-internet-guide.com/author/steve/.

[82] Jehoon Sung. "The Fourth Industrial Revolution and Precision Agriculture". In: *Automation in Agriculture: Securing Food Supplies for Future Generations*. BoD–Books on Demand, 2018, pp. 3–15.

[83] Circuito Team. *Everything you need to know about Arduino Code*. Online. [Last Accessed: January 2020]. 2018. URL: https://www.circuito.io/blog/arduino-code/.

[84] Korbin S Van Dyke, Paul Campbell, and Don Alan Van Dyke. *Computer for Execution of RISC and CISC Instruction Sets*. US Patent 7,047,394. 2006. URL: https://patents.google.com/patent/US7047394B1/en.

[85] Ovidiu Vermesan et al. “Internet of Things Strategic Research Roadmap”. In: *Internet of Things- Global Technological and Societal Trends* 1.2011 (2011), pp. 9–52.

[86] Liberios Vokorokos et al. “A Multicore Architecture Focused on Accelerating Computer Vision Computations”. In: *Acta Polytechnica Hungarica* 10.5 (2013), pp. 29–43.

[87] Eric Von Hippel. “Learning from Open-Source Software”. In: *MIT Sloan Management Review* 42.4 (2001), pp. 82–86.

[88] *W3Schools Online Web Tutorials*. Online. [Last Accessed: February 2019]. URL: https://www.w3schools.com/.

[89] John E Walsh. “Soil moisture sensor”. US Patent 4,540,936. 1985. URL: https://patents.google.com/patent/US4540936.

[90] *Water Situation in South Africa*. Online. [Last Accessed: June 2019]. URL: http://www.waterwise.co.za/site/water/environment/situation.html.

[91] Wei-Meng Lee and Clarence Chng. *Introduction to IoT Using the Raspberry Pi*. Online. [Last Accessed: October 2019]. 2020. URL: https://www.codemag.com/Article/1607071/Introduction-to-IoT-Using-the-Raspberry-Pi.

[92] *What is a Raspberry Pi?* Online. [Last Accessed: January 2020]. URL: https://opensource.com/resources/raspberry-pi.

[93] *What is an Arduino?* Online. [Last Accessed: January 2020]. URL: https://opensource.com/resources/what-arduino.

[94] *What is Open Hardware?* Online. [Last Accessed: June 2020]. URL: https://opensource.com/resources/what-open-hardware.

[95] *What is Open Source?* Online. [Last Accessed: January 2020]. URL: https://opensource.com/resources/what-open-source.

[96] *What is Publish/Subscribe? Is it Useful for Me*. Online. [Last accessed 15-January-2020]. URL: https://vernemq.com/intro/mqtt-primer/publish-subscribe.html#publish-subscribe.

[97] Dale Wheat. *Arduino internals*. 1st ed. Apress, 2012.

[98] Feng Xia et al. “Internet of Things”. In: *International Journal of Communication Systems* 25.9 (2012), pp. 101–1102.

[99] Min Xu, Jeanne M David, and Suk Hi Kim. “The Fourth Industrial Revolution: Opportunities and Challenges”. In: *International Journal of Financial Research* 9.2 (2018), pp. 90–95.

[100] Tomoaki Yamaguchi. *GitHub - ty4tw/MQTT-SN: MQTT-SN Gateway Client over XBee and UDP*. Online. [Last Accessed: February 2019]. URL: `https://github.com/ty4tw/MQTT-SN`.

[101] Ilaria Zambon et al. “Revolution 4.0: Industry vs. Agriculture in a Future Development for SMEs”. In: *Processes* 7.1 (2019), p. 36.

[102] Nikola Zlatanov. “Arduino and Open Source Computer Hardware and Software”. In: *International Research Journal of Engineering and Technology (IRJET)* 4 (2017).

Appendix A

Gateway Implementation Code

```
from digi.xbee.devices import XBeeDevice  # local device, connected
    to Pi3
from digi.xbee.devices import RemoteXBeeDevice  # remote node,
    connect to Arduino
from digi.xbee.devices import XBee64BitAddress
import paho.mqtt.client as mqtt

import time  # sleep services
import binascii # to convert bytearray to hexadecimal string, using
    hexlify

def actionHandler(client, userdata, message):
        print("in handler")
        node = message.topic[8:24] #fetch the node address from the
    message topic
        print("Received: action" + message.payload.decode("UTF") + "
    from server for node" + node )
        if node in nodes: #check if the node is known by the gateway
                    #if the node is known
```

```
        # put the action in dictionary for transmission when
    requested by node
            nodes[node] = message.payload.decode("UTF")
        else:
            #error if the node is not known
            print("! error, unknown node !")

client = mqtt.Client("Pi3-gateway3") #initialize an MQTT client
print("starting look")
client.connect("localhost") #connect client to local MQTT broker
client.loop_start() #to activate the callback functions
client.subscribe("/action/+") # subscribe client to all action
    topics
client.on_message = actionHandler #register the function to handle
    reception of MQTT messages
print("handler registered and active")

#create a local XBee device object on serial port
localDevice = XBeeDevice("/dev/ttyAMA0", 9600)
#open device connection
localDevice.open()
# just allow the connection to complete, just in case
time.sleep(2)


while True:
        print("listening to zigbee messages")
    #    block for 2 hours max;
    #       (a message will arrive before time out)
```

```
    readMessage = localDevice.read_data(7200) #read a ZigBee
message

 # get the MAC address of the sender
    remoteAddress = readMessage.remote_device.get_64bit_addr()
 # remoteAddress is a XBee64BitAddress object
 # remoteAddress is a property of the class XBee64BitAddress,
 # which contains the remote address in a bytearray;

 # tranform remote address into string
    remoteAddressAsHexString = binascii.hexlify(remoteAddress.
address)
 # hexlify transforms it into a hexadecimal string - byte string

 #  which must be decoded in order to be used with the MQTT
server
    remoteAddressAsSString =  remoteAddressAsHexString.decode("UTF
-8")

    # add the remote address to the nodes dictionary, if not there
    if remoteAddressAsSString not in nodes:
      nodes[remoteAddressAsSString] = "O"

    # the payload of the zigbee packet is a sequence of bytes;
 #  if the first byte is 0 then it is a 'report' (publish)
message
 #  if it is 1, then it is a request for action (subscription)
message

    #get the first byte of the payload byte array
```

```
    firstByte= readMessage.data[0]  # data is a bytearry

    if (firstByte != 0 and firstByte != 1):
    #message is neither a report nor a subscription
        print("***************>>>>>>>>>>>>>>>>>>>> read to clear")
   # then discard it

    if firstByte == 0:  # it is a report, publish
      print ("received: humidity report from node " +
remoteAddressAsSString)

      # prepare topic "/humidity/nodeMAC"
      topic = "/humidity/" + remoteAddressAsSString

      # get humidity value (0 to 1023)
      humidityBytes = readMessage.data[1:3]  # semantics of range!
      humidityValue = int.from_bytes(humidityBytes, byteorder = '
big')

      # pubblish!
      client.publish(topic, str(humidityValue)) #perform the
publish
      print("humidity published to the server in topic"+ topic)

    if firstByte == 1:
    # it is a request for action, retrieve action from nodes
dictionary
    # and send it to the node

      #first check if the node is known by the gateway
```

```
    if remoteAddressAsSString in nodes:
      # then fetch the node actuation from the dictionary
      action = nodes[remoteAddressAsSString]
      print("received: action request from node " +
                                remoteAddressAsSString)
      # send; the remote node is represented by the
remote_device,
      # which was loaded when the message was read
      try:
        #send the node action
        localDevice.send_data(readMessage.remote_device, action)
      except:
        print("Attention: not received!")
      else:
        print("sent: " + action + " to node "+
remoteAddressAsSString)
    else:
      print(" Error, node not in nodes dictionary !")
```

Sensor Node Implementation Code

```
#include <XBee.h>

//creating an XBee object
XBee xbee = XBee();

//Remote XBee device address
XBeeAddress64 addr64 = XBeeAddress64(0x0013A200, 0x417DE0CB); //
    coordinator on super node ( pi 3)
//create ZBTX and ZBRX objects
ZBTxRequest zbTx = ZBTxRequest(); // this will be used to transmit
    outgoing packets
ZBRxResponse zbRx; // this will be used to read incoming packets

void setup() {
  Serial.begin(9600);
  xbee.setSerial(Serial);
  //setting the zbTx remote device address to the coordinator
  zbTx.setAddress64(addr64);
  pinMode(13, OUTPUT);

}

uint16_t sensorValue = 999;
uint8_t MSB, LSB;

void loop() {
  sensorValue++;
  if (sensorValue > 1024) {sensorValue = 0;};
```

```
  MSB = sensorValue / 256;
  LSB = sensorValue % 256;
  uint8_t data[] = {0,MSB,LSB}; // 0 on the first byte means this is
    a publication
  zbTx.setPayload(data); // this is the actual message that will be
   contained by the transmission packet
  zbTx.setPayloadLength(sizeof(data));
  xbee.send(zbTx); //sending the publication to the pi3

  //We expect an acknowledgement packet after sending the
   publication
  xbee.readPacket(4000); // throw this one away
  delay(1000);  // wait 1 seconds before requesting  response of the
    controller
                //   (via sending a 'subscription' request)
// send subscritpion now
  uint8_t ndata[] = {1,MSB,LSB};  // 1 on the first byte means this
   is a subscription; proxy knows topic
  zbTx.setPayload(ndata);
  zbTx.setPayloadLength(sizeof(ndata));
  xbee.send(zbTx); //sending the subscription
  xbee.readPacket(4000); // throw this one away (acknowledgement
   packet)
  delay(2000);

// and listen for the response from the gateway
    xbee.readPacket(4000);  // surely the response will arrive
   within 4 seconds
//  load the packet into zbRx;
    xbee.getResponse().getZBRxResponse(zbRx);
```

```
//    now set LED depending on response from controller
    if (zbRx.getData(0) == 'O') {
          digitalWrite(13, HIGH);}
       else {
          digitalWrite(13, LOW);}

   delay(6000);  // wait 6 seconds before looping */
}
```

Local Client Implementation Code

```
import paho.mqtt.client as mqtt #import client
import paho.mqtt.subscribe as subscribe # for the callback
import time

nodes ={
        "0013a200417de439" :
        {
        "gps" : "",
        "crop" : "maize",
        "priority" : 3,
        "status" : "C",
        "etc" : ""
        },

        "0013a200417de0ba":
        {
        "gps" : "",
        "crop" : "maize",
        "priority" : 2,
        "status" : "C",
        "etc" : ""
        }
    }

def reportHandler(client, userdata, message):
      node = message.topic[10:27] #fetch the node address from the
   message topic
```

```
    if node in nodes: #checks if the node is known by the
controller
        print("working node action...")
        # run logic to determine what to do; here just a stub
        #load the node information from the dictionary of nodes
        info = nodes[node]
        action = info["status"] # and then fetch the node action
status valve "O" or "C"
        # publish the action to on the correct topic "/action/
nodeMAC"
        print("Received: node " + node + " humidity value " + action
)
        topic = "/action/" + node
        client.connect("localhost") # server running on this Pi
        client.publish(topic, action) #publish!
        print("published: node actuation to the local server")

        # change action, for easy testing of architecture
        if action == "O":
          action = "C"
        else:
          action = "O"
        # update the status in the dictionary with the info on the
processed node
        info["status"] = action
    else: # executes if the node is not in the dictionary of known
 nodes
        print("! error, unknown node !")
```

```
# let's register the callback, so that it receive all relevant
    messages
print("registering")
subscribe.callback(reportHandler, "/humidity/+", hostname="localhost
    ")
print("registered")
```

Database Store Implementation Code

```
from pymongo import MongoClient
import paho.mqtt.client as mqtt
import paho.mqtt.subscribe as subscribe  # to be used the register
    the callback
import time
import binascii
##


# function to handle message from localController, received via the
    MQTT broker
def actionHandler(client, userdata, message):
        # debug
        print("in handler")
            #extract the message category, the first character of the
    topic string
        category = message.topic[1]

        if category == "h":  # humidity
          # save the humidity value in the dictionary
                    # databased when an action arrives


                    # extract node address from the message topic string
                    # topic in this case is /humidity/nodeMAC (16
    characters long)
          node = message.topic[10:26]
                    #saving value in dictionary
          humidities[node] = message.payload.decode("UTF")
```

```
    if category == "a":  # action
      # add humidity and action to database

            #extract node address
      node = message.topic[8:24]

            #extract humidity value from dictionary
      humidityValue = humidities[node]

            #action value is carried by the message payload
      valveValue = message.payload.decode("UTF")

            #save values in the database
      collection = "_" + node # this is to 'parametrize' the name
  of the collection
      db[collection].insert({"humidity": humidityValue, "valve" :
  valveValue})

humidities = {}
# Connect to MongoDB - Note: Change connection string as needed
client = MongoClient(port=27017) # default port, on localhost
#    and create database
db=client.humidityAndValveValues2

#subscribe client and register callback function
subscribe.callback(actionHandler, ["/action/+" , "/humidity/+"],
  hostname="146.231.88.22")
```