DESIGN AND EVALUATION OF WIRELESS TECHNOLOGY SYSTEMS FOR DATA ANALYSIS

By

William Benjamin Van Der Merwe

Dissertation submitted in fulfilment of the requirements for the degree:

Master of Engineering

in

Electrical Engineering

Department of Electrical, Electronic and Computer Engineering

Faculty of Engineering, Built Environment and Information Technology

Central University of Technology, Free State

Supervisor: Prof. P.E. Hertzog

Co-Supervisor: Prof. A.J. Swart

March 2020

Declaration

I, William Benjamin van der Merwe, student number , do hereby declare that this research project, which has been submitted to the Central University of Technology Free State, for the degree: Master of Engineering in Electrical Engineering, is my own independent Work and complies With the Code of Academic Integrity, as Well as other relevant policies, procedures, rules and regulations of the Central University of Technology, Free State. This project has not been submitted before by any person in fulfilment (or partial fulfilment) of the requirements for the attainment of any qualification.

WB van der Merwe

13 March 2020

Acknowledgements

I thank God, my Heavenly Father, for giving me the opportunity, strength and ability to complete this thesis. I owe it all to You for providing me with knowledgeable people and colleagues who helped me throughout the journey to complete this master's thesis.

I wish to give my humble thanks and gratitude to my main supervisor, Prof P. E. Hertzog, for all the guidance and encouragement, and for providing me with the necessary technical suggestions during my research pursuit. I would also like to express my gratitude to my co-supervisor, Prof A. J. Swart for the useful comments, remarks and engagement through the review process. The doors to your offices were always open whenever I ran into a trouble spot or had a question about my research or writing.

Furthermore, I wish to express my sincere thanks to Michelle Schlechter for patiently helping me with MS Word and the setup for my thesis. To Herman Rossouw, I wish to express a special word of thanks for helping me with C++ and to better my programming skills. To Gavin Stuart from Nicks Electrical, I would like to thank for supplying me with the electrical panel to house my project.

Finally, I must express my very profound gratitude to my wife Carina for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without you. Thank you.

Author

Wimpie van der Merwe

Dedication

I dedicate this project to God Almighty my creator, my source of inspiration, my pillar of wisdom, knowledge and understanding. He has been the source of my inspiration, strength and determination. I also dedicate this work to my parents who have always encouraged me to study and taught me “No one can take knowledge away from you”. To my loving wife Carina, who has encouraged me all the way and who has made sure that I give it all it takes, to finish what I have started even when I did not believe in myself. Throughout my studies, you have been my anchor. Vlooi, this is for you.

Abstract

The internet and cloud storage are becoming increasingly important to researchers, hobbyists and commercial developers. This includes the transmission of reliable data as the availability and functionality of remote sensors and IoT devices are becoming more common. The availability of high-speed internet connections, like fibre-optic cable, LTE and digital radios, changed the playing field and enabled the user to transmit data to cloud storage as speedily as possible.

With these various technologies available, the question now arises: Which technology is more reliable and efficient for IoT sensors and for users to transmit data to a cloud server? This project aims to investigate the reliability and transmission delay of transmitted data from Wi-Fi, GPRS Class 10, and digital radio networks to cloud storage. A sampling unit was designed to evaluate analogue inputs periodically and send the recorded data to the three technologies under test. It also records the data to an on-board micro SD card along with an indexing system. The systems then transmit the sampled data and index number to a cloud storage server via the communication technologies under test. The cloud-stored data is then compared with the recorded data of the sampler unit to determine data integrity.

The transmission delays can be calculated by using the cloud storage server's time stamp information and the original time stamp of each data message. From the results acquired in the research, it showed that digital radio is a very reliable and stable means of data communication but it lacks direct connection to the internet. Although, both Wi-Fi and GPRS Class 10 are permanently connected to the internet, it was also observed that Wi-Fi internet connectivity may be susceptible to interference from external factors like the continuity of supply from the national power grid and from load shedding. It also showed that the XBee digital radio system lost 0.21% packets compared to the 0.31% for Wi-Fi and 1.46% for GPRS Class 10. On the other hand, although GPRS Class 10 may be a bit less reliable than digital radio and Wi-Fi, it is relatively cheap to use and has the ability to connect to multiple communication towers for communications redundancy.

The outcome of this research may help researchers, hobbyists and commercial developers to make a better-informed decision about the technology they wish to use for their particular project.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AC	Alternating Current
ADSL	Asymmetric Digital Subscriber Line
DC	Direct Current
GPRS	General Packet Radio Services
GPS	Global Positioning System
GSM	Global System for Mobile Communications
IoT	Internet of Things
ISP	Internet Service Provider
LTE	Long Term Evolution
MODBUS	Standard communication protocol connecting industrial electronic devices
NMEA	National Marine Electronics Association
OSI	Open System Interconnection
PV	Photo Voltaic
SD	Secure Digital
SFD	Save the Following Data
SPI	Serial Peripheral Interface
TCP/IP	Transmission Control Protocol/Internet Protocol
TF	TransFlash
TS Date	Time Stamped Date
TS Time	Time Stamped Time
VDC	Volt Direct Current
Wi-Fi	Wireless Fidelity
XCTU	XBee Configuration & Test Utility Software

CHAPTER 1: BACKGROUND AND INTRODUCTION

1.1 Background

The past 20 years have seen humanity progress from early internet-enabled PC's to modern mobile communication devices, to individual devices that can connect to the internet. The Internet of Things (IoT) allows one to register many devices via the internet and to pass information to a cloud server [1]. The essence of these IoT devices is the ability to create a path for data to flow from strategic locations to a cloud server. By using this method, researchers can interact with various locations, thereby obtaining relevant information to better understand various systems and environments. Cloud storage provides researchers with the ability to access accumulated data from any platform, anywhere in the world, with the added advantage that the data is always backed up [2].

With most common engineering research projects, researchers use electronic transducers and sensors to measure the results of their experiments. In some cases, numerous sensors are used for the outcome. This can provide unsafe working environments, as a vast number of sensor cables laid haphazardly over a specific area may cause injury. Other problems with cables are the tendency to break at critical times when optimum performance is required. The impedance of long cables also affects the accuracy of sensors, while deteriorating in corrosive environments. To overcome this problem, the researcher can utilise some type of wireless technology to connect sensors or transducers to the internet [3].

1.2 Problem Statement

With the development of new intelligent transducers and sensors, the trend is to connect these devices to the internet. With special websites available, these devices can store their sampled data online for later use. Lately, more and more cloud storage services are available for students or researchers to store sensor data, like ThingSpeak. Thus, the researcher can access the sample data from these devices from anywhere at any time. Taking this into account, the problem arises in determining the most cost-effective and

reliable way to transfer sampled data from a strategic location to an IoT server, like ThingSpeak, to enable data integrity and analysis.

In this study, three possible solutions will be investigated and evaluated, namely:

- Digital Radio;
- Wi-Fi; and
- GPRS Class 10.

1.3 Research Questions

With the infinite amount of communication technologies and internet connections present, the research explored the following questions:

- What technologies are available to transfer data via the internet to a cloud server for storage?
- Which technology is most reliable?
- Are there significant time delays between the technologies?
- Which technology is most cost effective?

1.4 Aim of the Study

The aim of this research is to determine the most cost-effective and reliable way for researchers to transmit data from sensors at a strategic location, to an internet cloud server using various wireless technologies. These include digital radio, Wi-Fi and GPRS Class 10.

To identify the most preferred wireless technology will require the design and installation of a physical test system from where empirical results may be obtained. This test system will need to simulate a real-life situation. There are various factors that will play a role in the outcome of the research, leading to the following objectives:

- Formulate a system design and determine the overall installation and maintenance costs for each technology;

- Analyse the simplicity of each technology;
- Determine the ease of implementation and power consumption for each technology; and
- Assess the data integrity for each technology.

1.5 Methodology

The research will include investigation, planning, design and practical testing. The collected data will be used to determine what the best solution is or what combination of technologies might be the best to transfer IoT data to a cloud service for storage. The three technologies used in the research is XBee digital radio, Wi-Fi, and GPRS Class 10. Every communication unit will be tested separately with the same data at the same time. Figure 1 shows a proposed block diagram of the system.

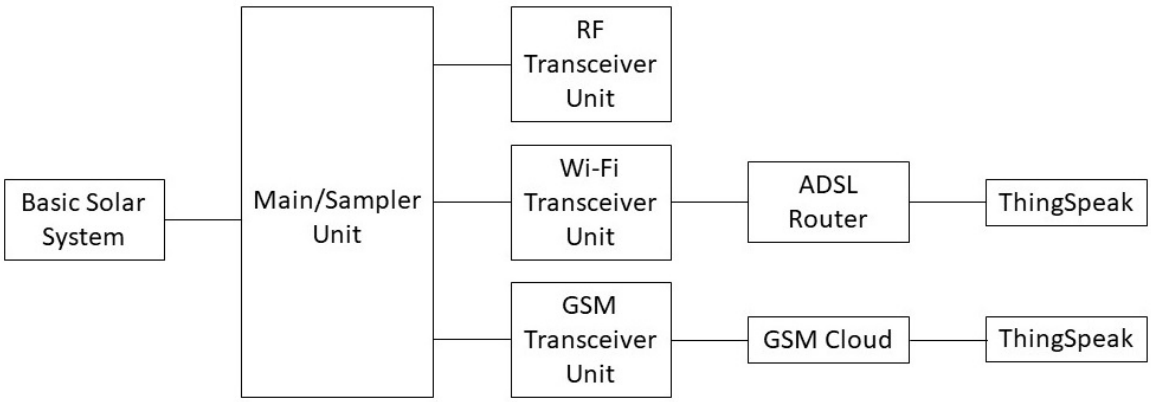


Figure 1: System Block Diagram

The purpose of the main or sampler unit is to continuously scan a basic photovoltaic solar system (or any energy system) for its voltage, current and temperature. It should receive the time information data from the on-board GPS that will be used for time verification throughout the system. This process should be a continuous loop to update the remote units. The main or sampler unit application will be designed in such a way that the system will send the analogue data sampled to a cloud storage server.

After a predetermined time, the sampler unit should indicate to the XBee radio, Wi-Fi and GSM units that the data that follows must be used for storage. The data received from the sampler unit should contain the time of the sample, unique index number and sampled analogue values. The unique index will then later be used to verify data from the different technologies under test.

Installation, maintenance costs, simplicity and ease of implementation for each technology will be determined over the evaluation period of the project. The power consumption for each technology will be determined by measuring the voltage and current values for each, in standby mode as well as transmit mode. The following results can be calculated by comparing locally stored data with remotely stored data:

- Propagation delay;
- Data integrity; and
- Percentage Data Packets Lost.

1.6 Benefits of the Research

The outcome of this research can give researchers, students and hobbyists a better understanding regarding the pros and cons of the different cloud storage wireless technologies offered, although there are many other options available. It may also promote the thought of a cableless environment by using wireless technologies.

1.7 Delimitations of the Study

The research will only focus on three major types of wireless communications, namely:

- XBee Digital Radio;
- Wi-Fi; and
- GPRS Class 10.

Furthermore, only the ThingSpeak cloud storage service will be used in this research. A simple photovoltaic system with battery storage and LED light for load purposes will be

used to provide measurable data. The exact design of this system does not form part of this research.

1.8 Outline of the Research

The dissertation is categorised into five chapters as follows:

Chapter 1: This chapter covers the background and introduction of the study. Firstly, this chapter highlights the problem statement, followed by the research questions and aims of the study. It also includes an overview of the research methodology, benefits and delimitations applied to achieve the required results of the study.

Chapter 2: This chapter reviews some of the available literature relevant to this field of study. Previous studies are analysed, considered and used as a guideline. The different components used in the research are also individually examined and discussed in this chapter.

Chapter 3: This chapter emphasizes the methodology of the research. It discusses the individual units and their functionality as well as methods used to obtain the required results. The communication protocol and the cloud storage environment on ThingSpeak are also reviewed.

Chapter 4: This chapter focuses on the data analysis and results obtained in the research, and has four main sections of results. Each section is discussed and described on its own and results are visually presented in the form of graphs, tables and screen captures.

Chapter 5: The final chapter focuses on the conclusion regarding the results obtained in the research. The discussion section debates the results obtained and the implications section deliberates on the various findings of the research. The conclusion summarises the combined outcome of the discussion and implications of the chapter.

1.9 Summary

Connecting intelligent transducers and sensors to the internet is a trend in modern society. A large number of online websites are available to support this infinite amount of data from these intelligent transducers and sensors. With this in mind, the researcher or hobbyist may wonder what is the most reliable and cost-effective way to store this information from transducers and sensors on the internet. This chapter gives an overview of these questions and proposed outline of the study. The next chapter discusses the literature reviewed for the various components and technologies used in this research. It also contrasts some of the key technical specifications of the individual components.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

The previous chapter focused on the need to determine the most reliable and cost-effective way to transmit transducer and sensor data to the internet for cloud storage. Technologies like XBee multi-hop communication, Wi-Fi and GPRS Class 10 can possibly be used to upload digital information to an internet cloud server for storage, to name just a few [4]. In this chapter, a literature review is done of possible renewable energy types which could be used as a data source for the research.

The capabilities of MODBUS communications protocol will also be reviewed for use as a possible serial data communication language. Then, the benefit of possible technologies like XBee digital radio, Wi-Fi and GPRS Class 10 will be considered to transfer IoT information to a cloud service for data storage. It is crucial to ensure that the same data is distributed to all the communication devices under test, sampled from the renewable energy data source.

2.2 Data Source

The primary location where data comes from is commonly known as source data or a data source. Thus, it is simply the source from where one can generate data that can be used in this research as input data to the three identified wireless communication technologies. Three possible renewable energy schemes will be considered as possible data sources. A renewable energy resource is any type of energy that can be restored or replenished to its original source during a lifetime. There are many forms of renewable energy systems currently available such as solar, wind power and biomass and many more, each of which have their own pros and cons [5][6]. Most of these renewable energies depend in one way or another on sunlight. Wind is a direct result of differential heating of the earth's surface which leads to air moving about, where biomass, on the other hand, is stored sunlight contained in plants [6] [7].

2.2.1 Solar Energy

The use of a solar renewable energy system as a viable data source is based on its defined measurable outputs. The charging voltage, current and solar panel surface temperature can be used as data source for the research. The only disadvantage is that the sun does not shine at night, which will affect the voltage and current readings, although the surface temperature of the solar panel will still be measurable.

Figure 2 below shows a typical commercial solar energy installation at De Grendel estate in Cape Town, South Africa. The solar panels are serially coupled to form arrays [8].

Figure 2: Commercial Solar Energy Installation at De Grendel Estate [9]

The sun is a fusion reactor that has been burning for over four billion years and provides us with free energy. In one minute, it provides enough energy to supply the world's energy needs for one year [10]. Horace de Saussure, a Swiss scientist, built the first solar energy collector in 1767, to heat water and cook food [11]. Considering that the first solar panels were made about 30 years ago, we have come a long way [12]. It is a fast-growing industry all over the world, with constant new developments in materials used and in the energy consumption required to manufacture these materials [13] [14] [15]. Solar powered energy is a non-polluting energy source, with no moving parts that could break

down, requires little maintenance, and has a life span of 20–30 years with low running costs [13].

Solar energy is especially unique as large-scale installations are not always required. Remote areas can easily produce their own electricity with installations varying from small to large depending on their needs. As the area grows and more energy is needed, more solar capacity can be added without having to replace installed components [13].

2.2.2 Wind Power

The use of wind as a renewable energy system and viable data source is based on its defined measurable outputs. The wind turbine can be used to charge a battery bank, where the charging voltage, current, wind speed and wind direction can be used as measurable data sources for the research. A disadvantage is that wind is not always constant at the geographical area where the research will be done.

Harvesting the energy of wind has been known to societies for thousands of years. The first known use of wind power was 3200 BC, when people used sails to navigate the Nile river [16]. The earliest written documentation comes from 1219 AD — the Cretans used sail-rotor windmills to pump water for crops and livestock [17]. Refinements to the windmill were mainly attributed to the Dutch in the early 1390s, who used multi-story towers with separate floors for living quarters, grinding grain, removing chaff and storing of grain [18].

Today, it's obvious that harvesting wind power is one of the most promising renewable energy sources [17]. In the early 1980s, wind turbines could generate 50 kW with a 15m rotor diameter standing 24m high. In 2012, a wind turbine can generate 10 MW with a 145m diameter rotor standing 162m high [19]. It is cost effective and environmentally friendly. According to the World Wind Energy Association, the global capacity wind turbines grew from 7 GW in 1997 to 372 GW by 2014, at an average rate of 21% per year [20].

Figure 3 shows the Gouda Wind Farm just outside Cape Town, South Africa. With 46 units operational, it is one of the largest wind-farms in Southern Africa generating 138 MW. The towers' hub height is 100m and commenced operation in September 2015 [21].

Figure 3: Gouda Wind Farm Outside Cape Town [21]

2.2.3 Biomass Energy

The term "biomass" refers to organic matter that has stored energy through the process of photosynthesis [7]. It can exist in one form as plant or animal material used for energy production [22]. Animal waste is also a good example — plant material is transformed through animal's digestive process into something humans can use for combustion [23]. Biomass is abundantly available on earth, and considered to be a common component for renewable energy production [24] [25]. Biomass can be transformed into various types of biofuels or energy by utilizing a number of processes. Some of these methods are thermal, physical, and biological processes [26] [27].

There are over 1,4 billion people that lack access to electricity, with 85 % of them in rural areas and the majority of them living in Sub-Sahara Africa. The number of people using traditional biomass as fuel is projected to rise from 2,7 billion today to 2,8 billion in 2030 [28].

A biomass renewable energy system can also be used as a viable data source with its own defined measurable outputs. Biogas can be produced from biomass, therefore the

temperature of the decomposing organic material can be measured as well as the amount and composition of the produced gas. The amount of organic material needed to produce measurable values can negatively impact the system and must be considered.

Figure 4 below shows a biomass plant in rural Germany. It takes only three people to feed and operate the plant, and can produce 700m^3 of biogas per hour. Overall, 7,8% of Germany's power generation and 11,7% of heat consumption were covered by bioenergy in 2015 [29].

Figure 4: Biomass plant in rural Germany [30]

Taking this into consideration, it was decided to use a solar renewable energy system as the data source for this research, due to the availability of the components and simplicity of the system. A pico-solar system is also economically viable and compact in size, requiring a much smaller installation area [31]. The voltage, current and surface temperature of a PV module in such a pico-solar system will be used for the data.

2.3 MODBUS

In order to transfer the measured data from the pico-solar system to the chosen wireless communication technologies, a communication protocol is needed. A communication

protocol is a set of rules that must be obeyed by all users in a communications network [32]. It specifies a common format so that all nodes know how to parse and construct data packets from each other. Data packets normally consist of three parts:

- Header;
- Data payload; and
- Footer.

The header contains control data such as a unique starting byte, addresses, sequence numbers, indication flags and message length. The data payload portion contains the user data of the message. It is usually made up of a type identifier, followed by a range identifier and then the actual data information. The footer of the message usually contains a checksum to validate the message [33].

Table 1 shows a typical MODBUS outline for request and response messages. Each data packet, whether it is a request or response, begins with the slave address, followed by a function code and parameters defining what is being asked for, or parameters provided to the slave. A checksum at the end of the message will verify the integrity of the data packet.

Table 1: Basic MODBUS Structure for Request and Response Messages

Address	Func Code	Reg Num	Reg Count	Data	CRC

A serial byte-orientated protocol like MODBUS is mostly used for communications over a point-to-point or point-to-multipoint link, defining it as a master/slave protocol [32]. It will always have one master device with at least one slave device. The device operating as a master (Main or Sampler unit) will poll one or more devices operating as a slave (XBee, Wi-Fi, and GPRS Class 10 units) [34]. This means that slave devices cannot offer information to the master, but must wait until the master requests information. The master device will write data to a slave device's registers and read data from the slave device's registers [32]. The master device will always stay in control of the data exchange request on the communications line, to eliminate cross talk. This is also known as balanced serial communications [35].

Previous studies showed that MODBUS is a “SCADA protocol that incorporates integrity, authentication, non-repudiation and anti-replay mechanisms” [36]. Other protocols, like DNP3.0 and IEC 60870-5-101, are much more advanced and complex than MODBUS, but not open to other encoding solutions [37].

MODBUS was chosen as the communication protocol for all the remote units in the wireless communication technologies, as it is an industry standard protocol used by most PLC’s and SCADA equipment [38] [39] [40] [41]. The protocol was created by Modicon (now Schneider Electric) in 1979. It remains the most widely available protocol for connecting industrial devices. The MODBUS protocol specification is openly published and use of the protocol is royalty-free [42]. On the other hand, hackers can easily infiltrate this communication protocol. However, it is of no concern in this study as the protocol is only used locally between the source data and the multiple slave devices, and not exposed to the internet.

2.4 XBee Radio Transceiver

To enable a master and slave device to communicate with each other using a given communications protocol requires the use of a specific wireless communication technology. The XBee digital radio is a wireless transceiver modem from Digi International that can provide communication for a robust communications network. Addressing, acknowledgements, and resending are standard features for safe delivery of data [43]. XBee modules are embedded solutions providing wireless end-point connectivity to devices.

These modules use the IEEE 802.15.4 [44] standard networking protocol for fast point-to-multipoint or peer-to-peer networking. They are designed for high-throughput applications requiring low latency and predictable communication timing. If there are two XBee’s in the same area, they will automatically 'sync' and pass serial data back and forth

without any additional work or configuration [45]. They are also very easy to set up and have error-correction techniques.

XBee modules have been used in various research projects for data monitoring purposes relating to temperature [45] and fault diagnosis of PV modules [46]. In the study done by the Sirindhorn International Institute of Technology in Thailand, multiple temperature sensors were monitored by a central master node polling the slave nodes on an hourly basis. The results showed that the system functioned satisfactorily.

In another study done by the Department of Electrical and Electronics Engineering, University College of Engineering, in Dindigul India, XBee transceivers were used to monitor the output current, voltage and irradiation of PV modules, along with the temperature and the irradiation of the environment. However, the level of data integrity was not reported on in this study, nor compared to other available wireless technologies. The uses for XBee digital radio are endless, though it is frequently used with IoT sensor devices that consist of nodes in a network and communicate with each other [47].

Some advantages of XBee digital radio include [43]:

- No wires are involved in the network;
- It can be set up as point-to-point, point-to-multipoint and mesh networks;
- Its low-duty cycle provides prolonged battery life;
- Low latency; and
- 65535 nodes per network.

Some disadvantages of the XBee digital radio:

- Limited range of communication;
- Risky to use private information;
- Prone to attack from unauthorized people;
- Low transmission rates; and
- It is not secure like Wi-Fi based systems.

Table 2 shows some of the basic specifications of the XBee Pro S2C [48]. It provides a simple RS232 communication interface for easy integration with many projects. What you send is exactly what the other modules get, and vice versa. It has an outdoor range of up to 3.2km @ 250kbps, and can be powered from most microprocessor boards, like the Arduino Uno, although it is not directly connected to the internet.

Table 2: Basic XBee-PRO S2C Specifications

Specification	**XBee-PRO S2C**	
Frequency	2,4 GHz	
Bandwidth	N/A	
Modem interface	RS232 serial interface	
Range	Indoor: 90 m	Outdoor: 3200 m
Transmit power (maximum)	63 mW (+18 dBm)	
Voltage	2,1 to 3,6 VDC	
Data rate (maximum)	Upload: 250 kbps	Download: 250 kbps
Receiver sensitivity	-101 dBm	
Adjustable power	Yes	
Operating temperature	-40 C to +85 C	
Deployment costs	Low	

2.5 Wi-Fi

XBee communication modules are very handy for point-to-point communications as well as point-to-multipoint communication. However, when internet connectivity is vital, then internet-coupled devices must be used. Wireless Fidelity, or better known as Wi-Fi, is a technology that uses radio waves to provide network connectivity without a physical wire connection [49]. A Wi-Fi connection is established using a wireless adapter to create hotspots in the vicinity of a wireless router that is connected to the internet, allowing multiple users or devices to access internet services simultaneously [50].

Once configured, Wi-Fi provides wireless connectivity to devices by emitting frequencies between 2.4 GHz–5 GHz [51]. A sensor device may include a wireless adapter that can translate monitored data into a radio signal. This same signal may be transmitted via an antenna to a decoder known as the router. The router will in turn send the received data to

the internet as specified by the sensor device [52]. The results of a study done at the School of Automation Science and Electrical Engineering, BeiHang University in Beijing, China, [53] showed that Wi-Fi has the following characteristics:

- High bandwidth;
- Non-line-of-sight transmission capacity;
- Large coverage area;
- Cost-effective;
- Easy expansion; and
- It is robust.

One key advantage of having Wi-Fi-enabled devices is that it allows for the seamless connection to Local Area Networks (and thus to the internet) [54]. Disadvantages that may occur are the vulnerability to security and the lack of reliability. In a study done by the Institute of Electronics, Information and Communication Engineers in Japan, it was reported that the recent popularity of IoT devices connecting to the internet via Wi-Fi are getting more and more attention.

Table 3 shows some of the basic specifications for Wi-Fi (IEEE 802.11g) [55]. It provides a simple RS232 communication interface for easy access. It has an indoor range of roughly 100m and an outdoor range of about 150m, resulting in a relatively small footprint. Initial deployment cost is high, but decreases after installation. The increase of more and more IoT devices using Wi-Fi networks creates great opportunities for attackers to perform malicious activities. Attackers impersonating themselves in Wi-Fi networks is one of the most serious threats, as they can disguise their presence to look like a legitimate IoT device [56]. Furthermore, it has been reported that the main limitations of Wi-Fi hardware include high power consumption and complex infrastructure requirements [57].

Table 3: Basic Wi-Fi Specifications

Specification	Wi-Fi	
Frequency	2,4 GHz / 5 GHz	
Bandwidth	20 MHz	
Modem interface	RS232 Serial interface	
Range	Indoor: ~100 m	Outdoor: ~150 m
Transmit power (maximum)	100 mW (20 dBm) on 2.4 GHz & 200 mW (23 dBm) on 5 GHz	
Voltage requirement	12 VDC	
Data rate (maximum)	Upload: 54 Mbps/450 Mbps	Download: 54 Mbps/450 Mbps
Receiver sensitivity	-65 dBm	
Adjustable power	Yes	
Operating temperature	-40 C to +85 C	
Deployment costs	High	

2.6 General Packet Radio Services

Wi-Fi can be very limited due to the area of coverage and infrastructure needed, where mobile communications can be implemented without any geographical constraints. Mobile communications is one of the fastest growing platforms ever. The number of mobile subscriptions are increasing globally, daily [58]. General Packet Radio Services (GPRS) is the first evolutionary step in deploying a truly mobile packet-based wireless communication service that promises direct internet connection for mobile phones and computers [59]. It can provide idealised data rates between 56 and 114 kB/s [60].

Mobile stations or wireless units routinely use GPRS as a packet data internet connection [61]. It has also been used for monitoring dispatch information of trucks and shovels in an open-pit mine [62], in vehicle-tracking systems [63] and in automatic meter-reading systems [64], to name a few. A study done by the Faculty of Electrical Engineering, University of East Sarajevo, in Bosnia and Herzegovina on "A Proposition of low-cost Sensor-Web implementation based on GSM/GPRS services" [65] concluded that GPRS could be used as a low-cost remote monitoring system. Another study done by the Beijing Forestry University in Beijing, China, confirmed this finding [66]. Both studies also reported that sensor information could be provided and accessed by the user on a

global scale.

Some of the advantages of GPRS:

- Always connected [67];
- Provides communication to remote nodes [66];
- Easy installation [66]; and
- Low cost [66].

Some of the disadvantages of GPRS [68]:

- Technical issues;
- Lack of support;
- Possible downtime; and
- Locking yourself to one supplier.

Table 4 shows some of the basic specifications for GPRS Class 10. It has the same RS232 communication interface as XBee and Wi-Fi, providing for simple easy access. The major improvement over XBee and Wi-Fi is that the range to the nearest communication node can be several kilometres, inside or outside. The upload and download data speed is relatively low compared to XBee and Wi-Fi.

Table 4: Basic GPRS Class 10 Specifications

Specification	**GPRS Class 10**	
Frequency	850 MHz/900 MHz/1800 MHz/1900 MHz	
Bandwidth	N/A	
Modem interface	RS232 Serial interface	
Range	Indoor: Several km	Outdoor: Several km
Transmit power (maximum)	2 W @ 900 MHz / 1 W @ 1800MHz	
Voltage requirement	3,2 to 4,8 VDC	
Data rate (maximum)	Upload: 42,8 kbps	Download: 85,6 kbps
Receiver sensitivity	< -106 dBm	
Adjustable power	Yes	
Operating temperature	-40 C to +85 C	
Deployment costs	High	

2.7 Summary of the three technologies

XBee, Wi-Fi and GPRS Class 10 each have their own advantages and disadvantages. XBee has some influential features over the high data rate of Wi-Fi, such as a wider coverage area, lower power consumption, large number of child-node connections, and better data encryption [69]. GPRS Class 10, on the other hand, is good for standalone sensors with small amounts of data.

Table 5 and Table 6 show a side-by-side comparison of all three technologies discussed in this chapter. XBee has the most nodes per network with a large footprint compared to Wi-Fi, where GPRS Class 10 has more standalone device support. The frequencies used are similar for all three technologies, with the exception of Wi-Fi at 5 GHz.

Table 5: XBee, Wi-Fi and GPRS Class 10 Comparison

Wireless Technology	Data Transmission Rate	Distance Coverage (approximate)	Frequency	Nodes per Network	Security
XBee	20 - 200 kbps	100-3200 m (45 km LOS)	868 MHz, 900 MHz 915 MHz and 2,4 GHz	65535	128-bit AES
Wi-Fi	1 Mbps to 54 Mbps	100 m to more	2,4 GHz and 5 GHz	> 1000	WEP, AES
GPRS Class 10	85,6 kbps Downlink 42,8 kbps Uplink	Up to 35 km	824 - 894 MHz / 1900 MHz	N/A	Proprietary

XBee is one of the most commonly used transceivers with a low data rate and power consumption. It has a wider coverage area, depending on indoor and outdoor use, that can vary from 100m to 45km depending on the model. It is ideal for developing prototypes and research-related activities. GPRS Class 10 allows the user to monitor and control sensors and automated systems around the world at any time.

The only drawback of GPRS Class 10 is the huge latency it could have, hampering real-time monitoring [70]. Wi-Fi, on the other hand, has a fast connection speed with a coverage

of about 100m, but is dependent on additional infrastructure to connect to the internet [71]. XBee is the cheapest option compared to the other two technologies but requires a third-party connection to the internet. It is also the easiest to implement in a simple system [72].

Table 6: XBee, Wi-Fi and GPRS Class 10 Comparison

Wireless Technology	Cost	Speed	Real Time	Complicity	Ease of Use	Power Consumption	Latency
XBee	Low	Medium	Yes	Simple	Easy	Low	>0.1 Sec
Wi-Fi	High	High	Yes	Complicated	Medium	Medium	4 Sec
GPRS Class 10	High	Slow	No	Very Complicated	Hard	High	8 Sec

2.8 ThingSpeak

Once an IoT device has connected to the internet via XBee, Wi-Fi or GPRS Class 10, it needs to store its data to some kind of storage platform. Cloud computing and utilization is becoming the long-term vision for computing, where data owners can remotely store their data on a cloud server and enjoy on-demand access to their data from anywhere in the world [73]. By utilizing the internet as the backbone of the communication system, objects and people can interact with each other, and a cloud server forms a critical part of this interaction.

Some of the more well-known cloud storage service providers include:

- Google Cloud IoT Core;
- AWS IoT Analytics;
- ThingSpeak;
- Axonize;
- Predix;
- AT&T IoT Platform;
- Oracle Internet of Things Cloud;

- SQLstream; and
- VMWare Pulse IoT Center.

Although there are a number of cloud service providers currently emerging , supporting a large number of IoT based devices [74], it was decided to focus only on ThingSpeak. This is mainly due to its simplicity, available examples and its ability to let the user visualize and analyse live data streams in the cloud.

The developers of ThingSpeak created this platform as an open-source IoT application to store and retrieve data [75]. ThingSpeak provides a long-term facility for researchers to store their data from sensors during logging applications, while also providing an instant visualization of the data to the researcher [76]. It is very easy for the user to configure sensor devices to send their data to ThingSpeak using IoT protocols. The stored data can be viewed live from a standard web browser or on a mobile device supporting Android or IoS. Data can also be downloaded in a CSV or JSON format from your domain for offline analyses [77].

In a study done on “Temperature and Heart Attack Detection using IoT (Arduino and ThingSpeak)” by the Adama Science and Technology University, Adama, ThingSpeak was used to record patients’ heart rate and temperature information. The doctor can set the periodic health check on the machine and the machine will then send the sampled information to ThingSpeak. The results of the periodical health check will then be sent to the doctor through ThingSpeak [78].

In another study featuring a “Smart Community Monitoring System using Thingspeak IoT Plaform” done at the Department of Electronics and Communication Engineering, Jyothi Engineering College, Kerala India, three homes were monitored by multiple sensors to monitor the environment and security conditions. The values of the sensor data were used to control devices or trigger an alarm. The data was sent to ThingSpeak where it could be visualised in charts and trigger various activities like sending alert messages via Twitter to the homeowners. The research concluded that the system was able to monitor and control

the homes as expected and that the homeowners could receive data from ThingSpeak [79].

The website was originally launched in 2010 as a service for IoT applications. The core element of ThingSpeak is the 'ThingSpeak Channel'. A channel stores the data that is sent to ThingSpeak with the following basic elements [80]:

- 8 fields for storing data of any type — these can be used to store the data from a sensor or from an embedded device;
- location fields — can be used to store the latitude, longitude and the elevation. These are very useful for tracking a moving device; and
- 1 status field — a short message to describe the data stored in the channel.

2.9 Summary

In this chapter, the literature review was presented. Firstly, renewable energy sources were discussed to find a suitable data source for this study, followed by the communication protocol used. Thereafter, each wireless communication technology used in this study were described with a summary, contrasting some of the key specifications shared amongst them. Lastly, the type of cloud storage service used in this study was presented. The next chapter discusses the practical setup of this study and all the individual technology units used for storing IoT data to a cloud service.

CHAPTER 3: RESEARCH PRACTICAL SETUP

3.1 Introduction

Chapter 2 was committed to presenting the literature review of the study. The literature review covered the actual hardware of the study and how they fit together to form the experimental system. Preceding studies were presented that highlighted applications of the hardware relevant to the current study. This chapter consists of the practical setup that includes the layout of the various components, how they fit together, communication protocol and the software with the different Hayes AT commands to make connectivity and data transfer possible to ThingSpeak via the internet.

The sampling unit stores the periodically sampled data to its own on-board storage, while the Wi-Fi unit and GPRS Class 10 unit saves the sampled data to a cloud service for storage. At the same time, the XBee radio will send the same sampled data to an XBee storage unit via a radio link. Due to the sequential execution of the program in the Arduino microprocessor, the microprocessor can only complete one task at any time; thus it cannot respond to polls from the master unit while it is transmitting data to the cloud service.

A failure to respond to the master unit's polls can be time tagged and this information, together with the stored data from each technology, can then be compared with each other and with the original source data from the sampling unit for validity, integrity and propagation delays. It is important to note that propagation delay is defined in this study as the time the slave unit takes to respond to the master unit after a successful event poll.

The practical setup can be divided into various components:

- Solar system (data source);
- Main sampler unit and RF transceiver;
- Radio transceivers with storage capabilities;
- Wi-Fi transceiver unit; and
- GPRS Class 10 transceiver unit

3.2 Project Design

The three wireless technology (XBee radio, Wi-Fi and GPRS Class 10) systems were designed and built with standard off-the-shelf electronic equipment. Each wireless system is based on an Arduino Mega 2560 microcontroller, along with each system's particular shields and supporting components, as per technology under study.

Cyclic measurements of the output voltage, current and temperature of a PV panel will be taken every 5 minutes. A period of 5 minutes will return 12 samples per hour, resulting in 288 samples per day. It will provide sufficient data to enable analysis, and to reach a reliable conclusions. Figure 5 presents the block diagram of the practical setup. Note that references to the diagram will be done in a matrix format, e.g. section 3C equals the "Transmitter Unit". Section 2 represents the master unit, section 4A the XBee slave unit, section 3B the Wi-Fi slave unit and section 3C the GPRS Class 10 slave unit.

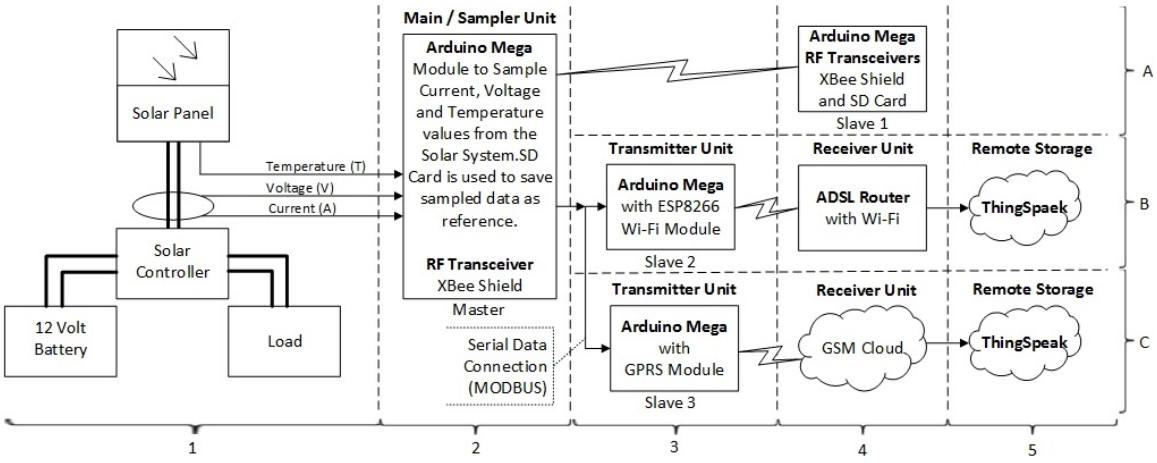


Figure 5: Block Diagram of Project

Section 1 of Figure 5 shows a 35 W solar panel, connected to a solar-charge controller. The solar panel and solar-charge controller will charge a 12 volt 7.2 Ah battery during the day. The battery will be drained during the night with the help of a static load. To create measurable results for the sampler unit, a voltage and current sensor will be connected to the solar panel. The surface temperature of the solar panel will also be measured.

The sampling unit is used to evaluate the analogue inputs periodically, sending the sampled data to the three wireless technologies under test. The sampling unit also records these sampled values to an on-board logger to keep track of the values using an indexing system. The index system is used to evaluate data integrity of all three wireless technologies under test.

The slave units connected to the internet will then send the sampled data and index to the cloud for storage via their individual communication technologies, while the radio unit will save its data locally to a remote SD-card unit via the radio link. The cloud-stored data can then be downloaded for comparison with the recorded data of the sampler unit in terms of correctness, using the indexing system and time-stamp information. The data from the XBee radio unit can be verified by the recorded data on the SD card.

The Sampler Unit in Section 2 of Figure 5 consists of an Arduino Mega 2560 microcontroller, equipped with a GPS shield and SD card for data storage, and an XBee shield and radio. The data values stored on the Sampler Unit SD card will later be used as reference data for the stored values of Section 4A, Section 5B and Section 5C, to calculate the validity, integrity and propagation delays of the stored data. The sampler unit will also forward the sampled data to the Arduino Mega microcontrollers in Section 4A and Sections 3B and 3C via industry standard MODBUS serial data communication protocol. The microcontrollers of Section 3B and 3C is hard wired to the sampler unit, while the microcontroller in Section 4A is connected to the sampler unit via XBee radio.

The XBee radio unit (Figure 5 Section 4A) uses an Arduino Mega 2560 microcontroller with an XBee shield on top of it as well as a SD card data logging shield for data storage. The 2 XBee radios are set up to communicate with each other in the same network. When data needs to be stored, the Arduino microcontroller will access the SD card and append the newest information to the log file.

The Wi-Fi Transmitter Unit (Figure 5 Section 3B) uses an Arduino Mega 2560 microcontroller with an external ESP-01 Wi-Fi module. At start-up, the Arduino

microcontroller connects and authenticates itself to the internet-connected Wi-Fi router. When data needs to be sent for cloud storage, the Arduino microcontroller will create a connection to ThingSpeak via the Wi-Fi module using standard Hayes AT commands. Once connected to the cloud server, it can transfer the data and close the connection.

The GPRS Class 10 Transmitter Unit (Figure 5 Section 3C) uses an Arduino Mega 2560 microcontroller with a GPRS CLASS 10 module connected serially to comport 2. The GPRS CLASS 10 module will connect itself to the network of choice. When data needs to be sent for cloud storage, the Arduino microcontroller will establish an internet connection via the GPRS CLASS 10 module using standard Hayes AT commands. Once the internet connection is established, it will validate itself to the cloud server and send the sampled data to the cloud storage website. On completion of the data transfer, it will close the connection to the cloud service.

Figure 6 shows the practical layout of the cabinet with the master unit, slave units and cloud communication equipment. All the microcontrollers are housed in a single cabinet with a serial hard-wired connection which connects the microcontrollers to each other, except for the XBee radio unit which is connected to the sampler unit via digital radio.

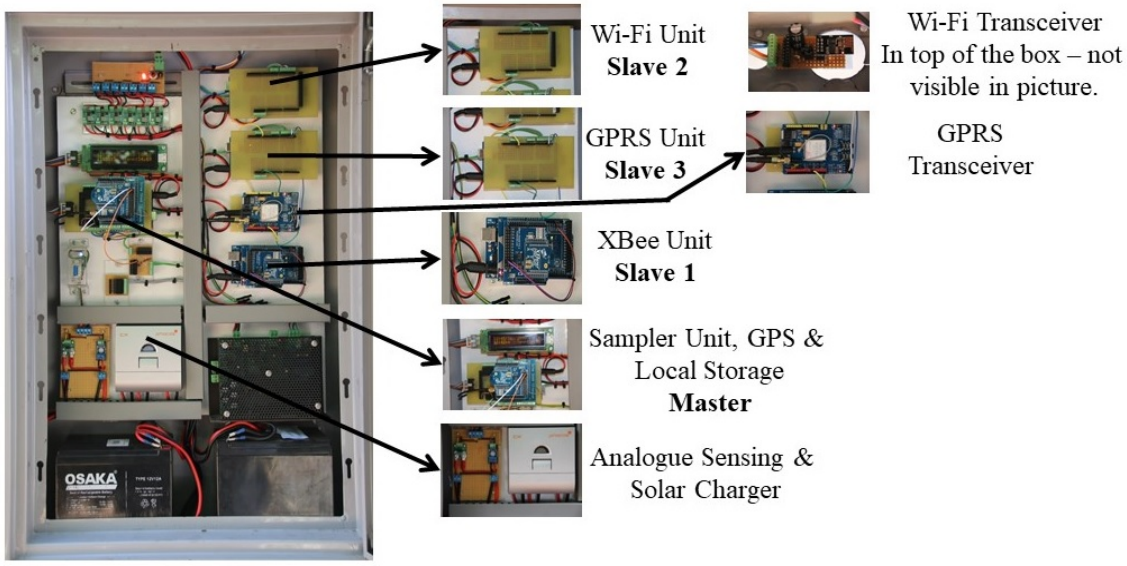


Figure 6: Practical Layout of Experiment.

The serial data communications speed is set to 19200 bits per second to secure quick and smooth data communications. The cabinet is also equipped with a battery charger and auxiliary battery to power the microcontrollers. This was done to ensure that the solar charger does not affect the power supply to the microcontrollers when the solar-charged batteries run low on charge.

3.3 Solar System Components

A simple solar system is used as a source to create measurable data for the study. The energy produced by the 35 W solar panel is fed via a coupled current sensor and parallel-coupled voltage sensor to the solar-charge controller. The output of the current sensor is connected to the first analogue input of the Arduino Mega microcontroller and the voltage sensor is connected to the second analogue input of the Arduino Mega microcontroller. There is also a temperature probe attached to the back of the solar panel with an aluminium plate for maximum heat transfer. The data output of the temperature probe is connected to I/O pin 2 of the Arduino Mega microcontroller. All the sensors are supplied with 5 volt from the Arduino Mega microcontroller.

The charge controller protects the battery from being overcharged by the solar panel and from being deep discharged by the load. The charging characteristics include several stages, which include automatic adoption to the ambient temperature. The charge controller decides when it is dark enough and switches the 10 W LED light on. The 10 W LED light will discharge the battery until the charge controller senses that the battery voltage is 11.4 V, and then disconnect the load to protect the battery from unwanted deep discharge.

The current sensor uses the ACS714 Hall-effect IC for the precise measurement of AC or DC current [81]. The device consists of a precise linear Hall circuit with a copper conduction path located near the surface. Applied current flowing through the copper path generates a magnetic field which the Hall IC converts into a proportional voltage.

The voltage sensor is a small module that uses a potential divider to reduce any input by a factor of 5. This reduction allows one to make use of the Arduino's analogue input to monitor the voltage of the solar panel. With a 5 V maximum input on the Arduino's analogue inputs, one is able to measure up to 25 V, which is ideal as the solar panel has an open-circuit voltage of 21.6 V [82].

A waterproof temperature probe is attached to the back of the solar panel with an aluminium plate for maximum heat transfer and provides 10-bit temperature readings over a 1-Wire interface. The 1-Wire interface is particularly handy for measuring something far away as the cable's length does not influence the temperature readings [83]. The temperature-to-digital converter chip is situated in the temperature probe itself, and only requires 1-line plus ground-to-return data signalling. Figure 7 shows the layout, sensor placement and sensor connections to the Arduino microprocessor for the solar system and sampler unit.

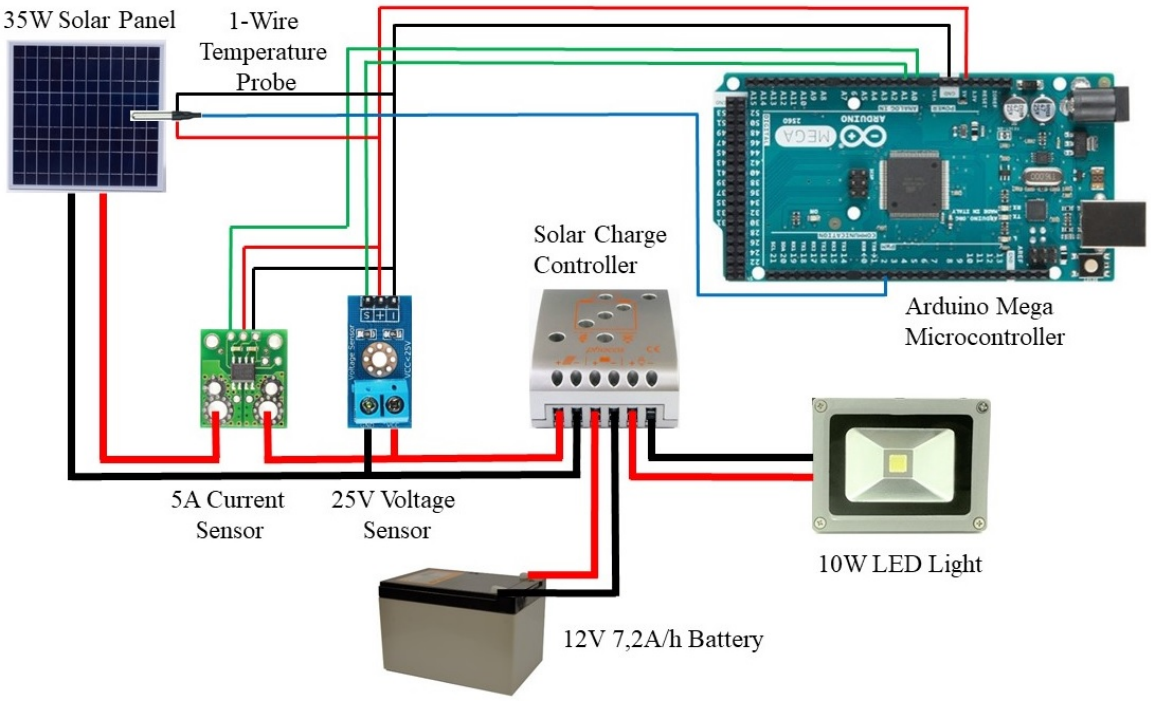


Figure 7: Solar System Components and Analogue Input Sensors

This figure is also the practical layout of Section 1 and Section 2 of Figure 5. The power supply of the microcontroller is not shown in Figure 7, as it does not form part of the study and is a power source on its own.

The Arduino Mega microcontroller continuously checks for analogue changes on the charging circuit of the solar system (see annexure A – line 317 and line 322) and writes the calculated current, voltage and temperature values to predefined registers for later use in the program. The microcontroller is responsible for calculating the analogue values, for reading time from the GPS and communicating to the other microcontrollers to update them with the latest information collected. These tasks will be discussed in detail in the following sections of this chapter.

3.4 Sampler/Main Unit

The Sampler or Main Unit is the core or the brain of the system. The unit gathers, stores and distributes all information needed in the practical setup. It scans the solar system continuously for any analogue value changes, updates its date and time every second from the GPS, builds a MODBUS data message from the collected data and transmits the data to the XBee, Wi-Fi and GPRS Class 10 units. It also decides when it is time for all these units under test to save their data. The XBee unit will then save its data to its local SD card while the Wi-Fi and GPRS units will save their data via their individual technologies to ThingSpeak for cloud storage.

The GPS shield is based on the Ublox NEO-6M GPS receiver and is pin-compatible to the Arduino Mega board. It is a well-known, well-performing complete GPS receiver with a built-in antenna that provides strong satellite search capabilities [82]. The shield also supports a Micro-SD card interface for data logging. It communicates serially at a speed of 9600 baud and sends NMEA sentences to the microcontroller to decode the time, date, latitude, longitude, altitude, estimated land speed, and fix type.

3.4.1 System Components

Bear in mind that the Arduino Mega microcontroller used for the analogue sampling of the solar system is the same microcontroller used for the GPS serial data decoding, serial data communications to the slave units and local information storing. More details on this will be discussed later in this chapter. Figure 8 shows the Sampler unit connected to the GPS module. Note that connections to other equipment are left out for simplicity of the diagram. The grey and blue wires represent the serial communications between the two circuit boards.

The GPS shield is pin-compatible and plugs into the Arduino Mega microcontroller, connecting all the necessary pins to power the GPS. Figure 8 (Serial Communications Pin Selector) shows how serial communications are user-definable to which pins are used for transmitting and receiving serial data.

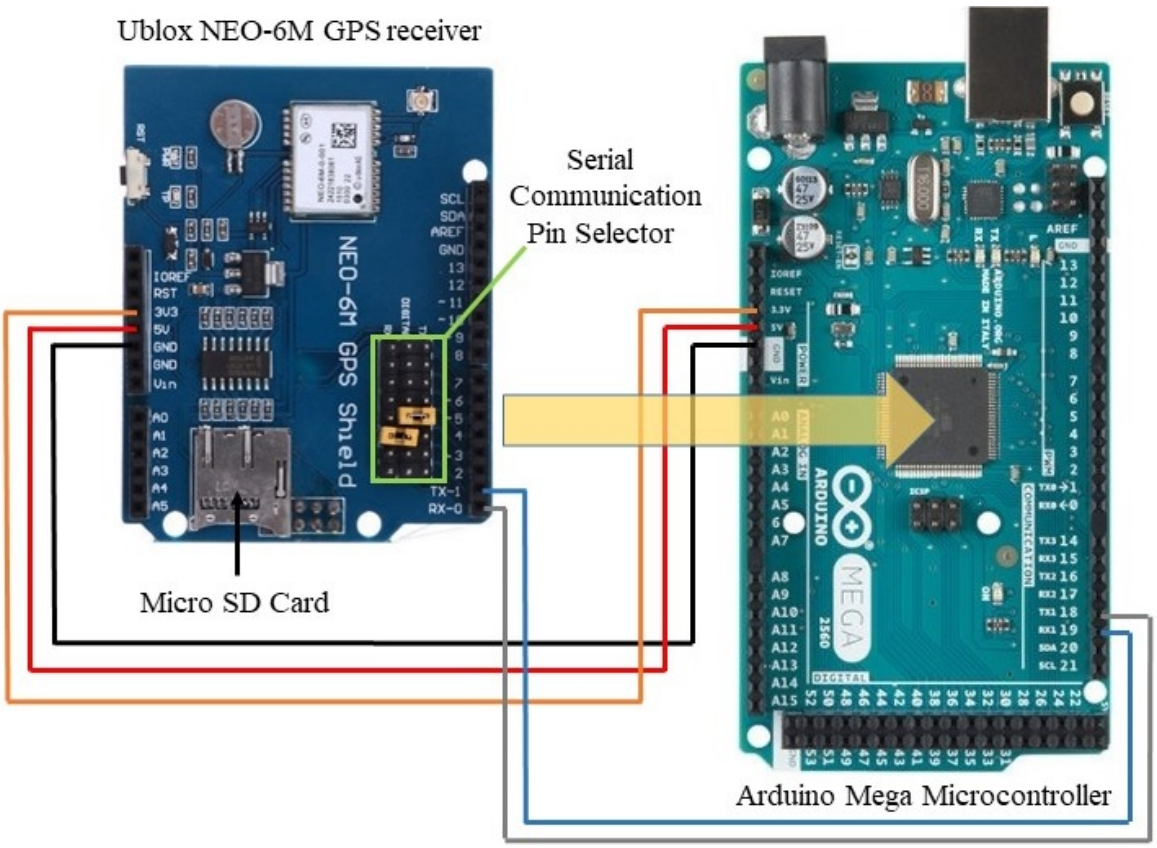


Figure 8: Sampler Unit with GPS shield

As a result of the microcontroller doing multiple communication tasks, it was decided to use the Arduino Mega microcontroller that supports four serial communication ports. The first communication port is used for program debugging, the second communication port is used for the GPS and the third communication port is used for information updates to the remote units using the MODBUS protocol.

The GPS transmits NMEA information (Figure 9) every second to the microcontroller. As the microcontroller's serial port is interrupt-driven, it starts to decode the received information the moment the interrupt is triggered (see annexure A – line 721). Once the information is decoded, the microcontroller updates the different holding registers (see annexure A – line 724) with the latest information.

Figure 9 shows typical data received from the GPS module. All of this information is presented to the user every second, cyclically. The message prefixes represent the following:

- $GPVTG - vector track and speed over the ground;
- $GPGGA - fix information;
- $GPGLL – latitude and longitude data;
- $GPGSA - overall satellite data;
- $GGSV - detailed satellite data;
- $GRMC - recommended minimum data for GPS; and
- $GPZDA - date and time.

A scheduled timer process (see annexure A – line 317) runs every 5 minutes in the main loop of the program. Once the timer kicks off, it sets an indication register (see annexure A – line 352) to a value of "1". This register indicates to the slave units that the information following must be used to update ThingSpeak via the technologies under test. This event is also used to write the information to the on-board Micro SD card of the GPS shield. This locally-stored information will later be used to compare the various data sets with each other to determine their data validity, integrity and propagation delays.

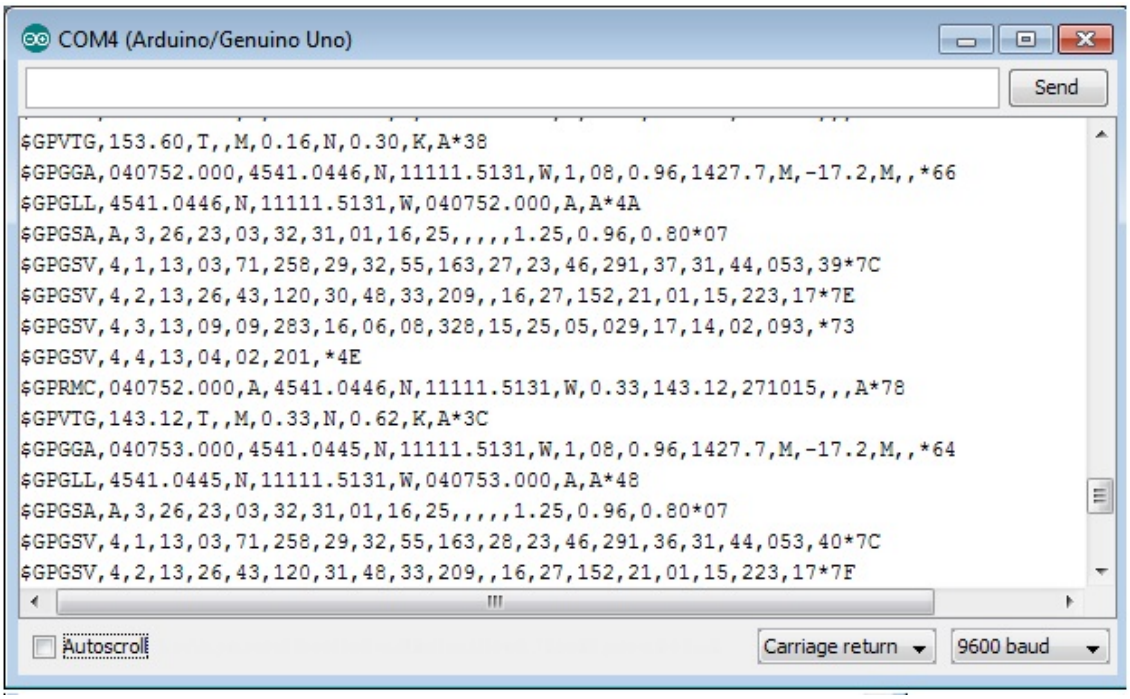


Figure 9: NEMA Data from GPS

An indexing system is used to keep track of every event stored locally and remotely. This index is a unique number for every event transmission to the technologies under test. At start-up, the program will open a file called “Index.txt” (see annexure A – line 220) and read the last value that was stored to it. The value read from the index file then sets the index integer value (see annexure A – line 232) in the program and is used as the new index value. As soon as the 5-minute scheduler process starts to accumulate information for transmission, the application advances the index value by 1 and only saves the new index integer value to the index file after a successful data transmission (see annexure A – line 410 to line 415). This process ensures that the system always uses a unique index number, even after a system restart.

The system also stores all the accumulated information to a local log file. The local log file is called “LogFile,txt” (see annexure A – line 260) and is updated (see annexure A – line 367 to line 414) with the latest information after each 5-minute period. The information captured in the main log file is later used for comparison with the remotely stored data to calculate data integrity and reliability.

Continuous information updates are sent to the slave units from the sampler/main/master unit. Communication port 2 of the Arduino Mega microcontroller is used to do this task using the industry standard MODBUS ASCII protocol [38]–[41] (the implementation of MODBUS is discussed later in this chapter). It is easy for the master unit to transmit to the multiple slave units as the slaves are linked together in a multi-drop fashion. If one of the Arduino's digital pins are set to a high-input impedance, it is equivalent to a series resistor of 1 MΩ in front of the pin. This high impedance allows one to connect the transmit pin of the master unit to the receiving input pins of multiple slave units. All the slave units receive the same data sent by the master at the same time, but only the unit with the corresponding address will respond to the message.

The slave units need to reply with an acknowledgement message to the master unit on every data poll from the master unit. This acknowledgement message notifies the master unit that the slave received the data message correctly and error-free, and that the master unit does not need to resend the message but can continue to the next slave unit. Figure 10 shows a block diagram of the hard-wired serial data connection from the slave units to the master unit. The 4700 Ohm resistor connected to VCC is used to reduce noise on the communication line from the slave units to the master unit. It will keep the communications line at a slightly "high" potential, but will not interfere with the data since RS232 communications work on an active "low" basis.

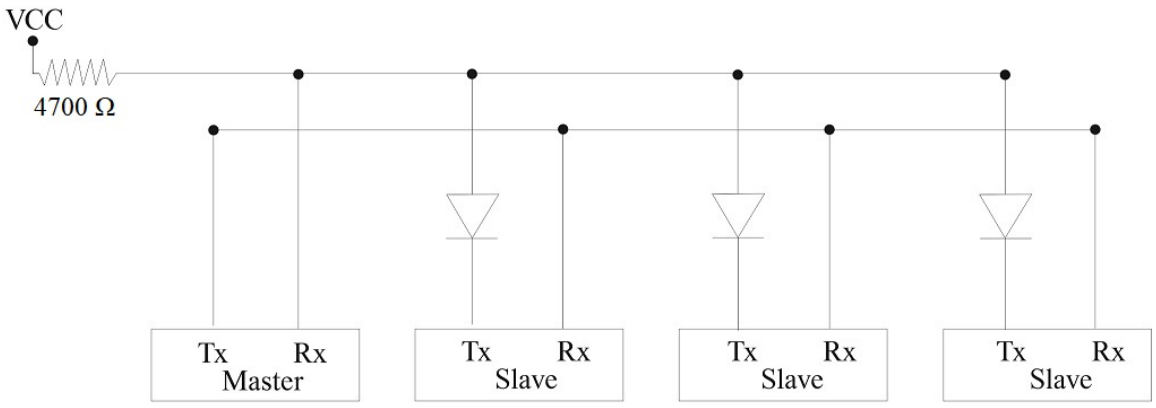


Figure 10: RS232 Data Communications Diode Configuration

The transmit path from the slave units to the master unit is a bit more complex. Keep in mind that the receive pin of the master unit is configured as an input and therefore in a high impedance state. The transmit pins of the slave units are connected to the master unit via a series diode to protect the units from damage if an adjacent unit is replying to the master unit after an information poll. The 4700 Ohm pull-up resistor is connected to 5 VDC to keep the TTL logic of the RS232 communications line slightly high when the communication line from the slaves to the master is in an idle state.

Figure 11 presents the Arduino Mega microcontroller with the additional GPS, screw and XBee shields, that will make up the stack on top of it. It is built up in layers with one shield stack on top of the next shield. The screw shield is only used for easy wire terminations.

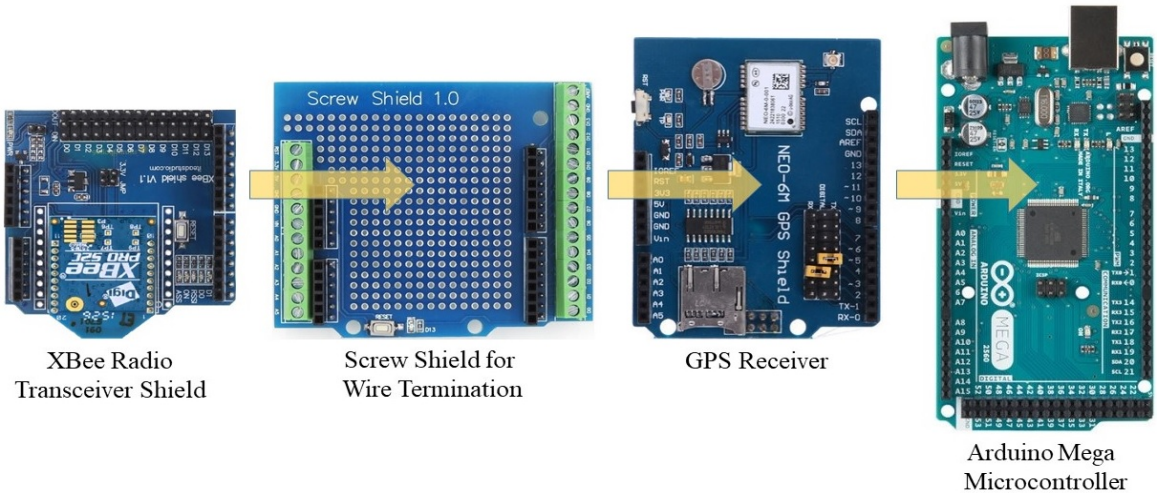


Figure 11: Various Shields stacked onto the Arduino Mega Microcontroller.

Figure 12 is a picture of the physical unit with the various shields stacked on top of the Arduino Mega microcontroller. It starts with the Arduino Mega at the bottom followed by a custom screw terminal board and then the GPS module. An additional screw terminal board, XBee shield and the XBee radio the follow. The XBee digital radio is only powered from the microcontroller and forms part of the MODBUS communications network.

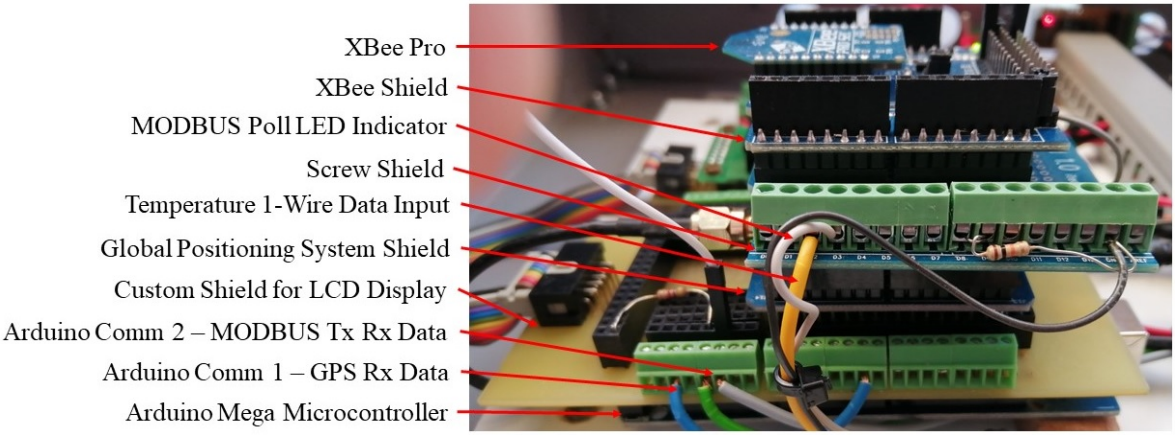


Figure 12: Master Unit with Shields Stacked

3.5 MODBUS

The most commonly used form of MODBUS protocol is MODBUS RTU. MODBUS RTU is a relatively simple serial protocol that can be transmitted via traditional UART technology like the Arduino Mega microcontroller. Data is transmitted in 8-bit bytes, one at a time at 19200 baud (bits per second).

A typical MODBUS RTU network has one master and one or multiple slave devices. Each slave has a unique 8-bit address. Data packets sent by the master device include the address of the slave the message is intended for. The slave will only respond when it recognizes its own address. The response from the slave must be in a certain time period or the master will detect it as a “no response” error.

Each data exchange request from the master is followed by a response from the slave. Each data packet, whether it is a request or response, begins with the slave address, followed by function code, followed by parameters defining what is being asked for or provided to the slave. A checksum at the end of the message will verify the integrity of the data packet. If the checksum does not calculate to the correct result, the slave will discard the message and the master will identify the data packet as a “no response”. The master will then retry the data packet for a predefined number of retries.

Table 7 illustrates the general outline of each request and each response MODBUS message. The device address range for devices varies between 0 and 255 and must be unique to each device. From a possible 255 function codes, only 8 are commonly used. The register number, register count and data value is made up of two bytes, giving them 65 535 possibilities. The checksum is two bytes added to the end of every MODBUS message for error detection. Every byte in the message is used to calculate the checksum.

Table 7: General Outline for each Request and Response Message

Device Address
Function Code
Register Number
Register Count
Data
Checksum

MODBUS messages are read and written as “registers” which are 16-bit pieces of data. These registers are known as Holding Registers and can be read or written. Another type of register is an Input Register, which is read-only. The function code will determine the type of register addressed by a MODBUS data packet. The most common function codes include type 3 “read holding registers”. Function code 6 is used to write a single holding register and function code 16 is used to write to one or more holding registers. Only function code 16 (Preset Multiple Registers) was used in this study. The raw data below illustrates a typical MODBUS request message from the master device to a slave device to pre-set multiple registers with values.

0x01 0x10 0x00 0x00 0x00 0x0C 0x18 0x00 0x00 0xE9 0x40 0x07 0xE3 0x00 0x04 0x00 0x1B 0x00 0x0B 0x00 0x3B 0x00 0x0C 0x07 0xBD 0xFF 0xD7 0x15 0xB4 0x00 0x00 0xA9 0xAD

Table 8 shows the MODBUS message from the master to the slave units in Hexadecimal

taken from the raw data message above. It can be seen that slave 1 is addressed by the master; the master requests the slave to "Write Multiple Registers" starting from address 1 and the following 12 registers will be updated. The resulting data pay load will consist of a total of 24 bytes.

Table 8: MODBUS message from Master to Slave

Description	Hex Value	Decimal Value
1st Byte - Device Address	0x01	01
2nd Byte - Function Code	0x10	16
3rd Byte – Address of the First Register – Hi Byte	0x00	00
4th Byte – Address of the First Register – Lo Byte	0x00	00
5th Byte – Number of Registers – Hi Byte	0x00	00
6th Byte – Number of Registers – Lo Byte	0x0C	12
7th Byte – Number of Bytes to follow	0x18	24
8th Byte – Value – Hi Byte	0x00	00
9th Byte – Value – Lo Byte	0x00	00
nth Byte – Value – Hi Byte	0x00	00
nth Byte – Value – Lo Byte	0x00	00
Checksum - CRC	0xA9	169
Checksum - CRC	0xAD	173

The raw data below illustrates a typical MODBUS response message from the slave device to the master device to a pre-set of multiple registers.

0x01 0x10 0x00 0x00 0x00 0x0C 0xC0 0x0C

Table 9 shows the MODBUS message from the slave to the master unit in Hexadecimal format from the raw data above. Slave 1 responds to the master, confirming that 12 registers were updated from the "Write Multiple Registers" request.

Table 9: MODBUS message from Slave to Master

Description	Hex Value	Decimal Value
1st Byte - Device Address	0x01	01
2nd Byte - Function Code	0x10	16
3rd Byte – Address of the First Register – Hi Byte	0x00	00
4th Byte – Address of the Second Register – Lo Byte	0x00	00
5th Byte – Number of Registers – Hi Byte	0x00	00
6th Byte – Number of Registers – Lo Byte	0x0C	12
Checksum - CRC	0xC0	192
Checksum - CRC	0x0C	12

Figure 13 below is a screen capture of the "ASE 2000" protocol analyser showing MODBUS protocol polling in motion, where the master unit polls all 3 slave devices in the practical setup of this study. This sequential polling of the slave devices will continue indefinitely. The red messages form the poll from the master unit to the slave unit and the blue message is the acknowledgement or reply from the slave unit to the master unit.

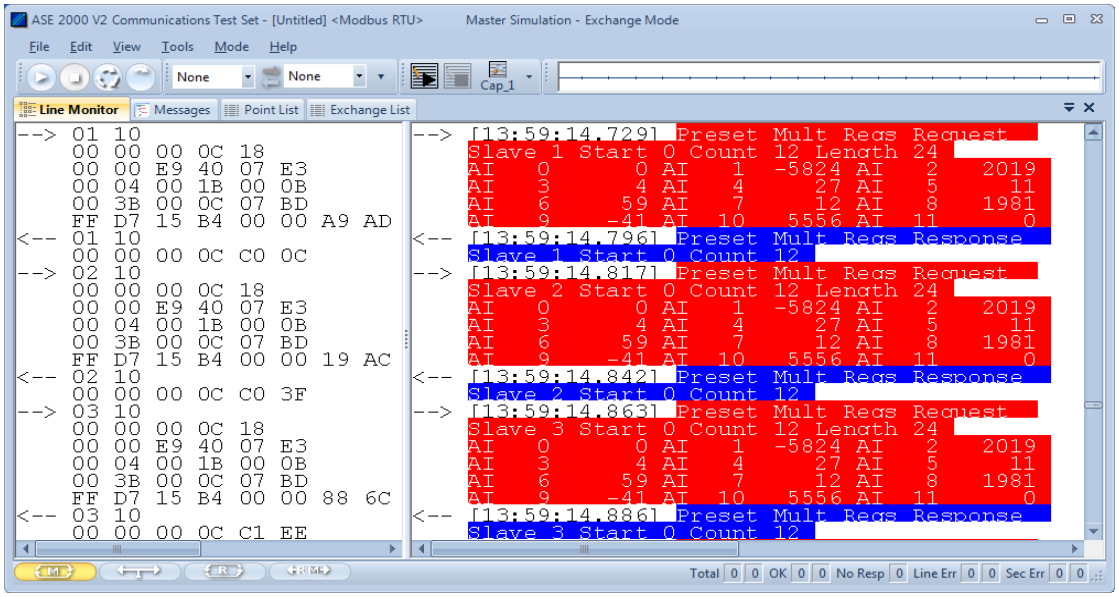


Figure 13: MODBUS Poll from Master and Response from Slave

If the 1st register (8th & 9th Byte) is set to a value of 1, it indicates to the slave device that the register data that follows must be used as valid data that must be stored or sent for cloud storage. In this study, it was decided that this particular byte will be known as the "Save the Following Data" (SFD) byte. Figure 14 shows an example where the 1st register is set, as an indication to the slave to store the following data. Note that AI 0 value is set to a value of 1.

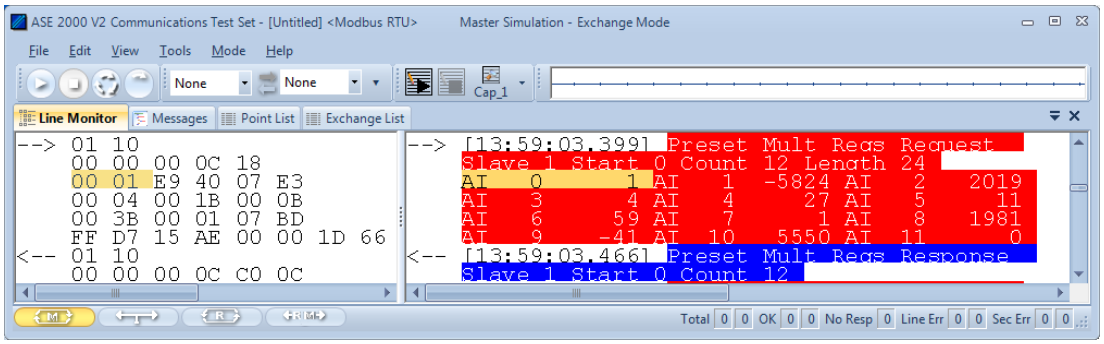


Figure 14: MODBUS Master setting first analogue value to 1, to indicate the data to follow must be sent for cloud storage.

Table 10 is an explanation of all the payload data registers that the master device can set on the slave devices for this study. The slave address, function code, register address, number of bytes and checksum is left out for simplicity. Register A0 indicates to the slave that the registers must be updated with the following data. This is true for all three slave devices on the communications line, to keep the data exactly the same for all of them.

Register A1 is the unique index number that will be used throughout the system to keep track of and synchronise all the data records that were saved locally by the master unit and the XBee unit as well as the remotely stored cloud data. Register A2 is the year, A3 is the month and A4 is the day as received from the GPS. Registers A5, A6 and A7 is a representation of the time as received from the GPS. Thereafter, register A8 is the value of the voltage and register A9 the value of the current from the solar system. Register A10 is the value of the surface temperature of the solar panel. Register A11 will indicate to the slave devices to do a soft restart, should the master unit decide that there is a

problem with the system and needs to reboot. A total number of 12 registers are addressed, making up the 24 bytes used as shown in table 8.

Table 10: Description of Settable Slave Data Register

Register	Description
A0	Indicates to slave device that the following data needs to be saved
A1	Unique index to for each data message that needs to be saved
A2	Year value from the GPS receiver
A3	Month value from the GPS receiver
A4	Day value from the GPS receiver
A5	Hour value from the GPS receiver
A6	Minute value from the GPS receiver
A7	Second value from the GPS receiver
A8	Voltage value of the solar system
A9	Current value of the solar system
A10	Temperature of the solar panel
A11	Remote soft restart for slave devices

3.6 XBee Slave Unit

The XBee slave unit consists of a SD Card Data Logging Shield, a XBee Shield with an XBee Transceiver Radio and an Arduino Mega microcontroller. The SD card shield supports a micro-SD card as well as a TF card. Both shields are pin compatible with the Arduino UNO and Mega microprocessors. The configuration of the XBee radios is done with proprietary software “DIGI XCTU”, supplied by DIGI free of charge and available for download on the internet. The full screen-captured configuration of both radios can be seen in Annexure E.

Communications port 0 of the microcontroller is used for software debugging, while communications port 1 of the microcontroller is used for MODBUS communication to the microcontroller via the XBee radio system. As the XBee shield is pin compatible with

the microcontroller, it is possible to configure some of the digital I/O pins as serial communication pins also known as “Software Serial”. Links on the shield must then be set to the corresponding digital I/O pins to transmit and receive as shown in Figure 15. This will then imply that one will have to make use of an additional library like “Software Serial”, to create a serial data connection on the software-defined digital I/O pins of the microcontroller.

The use of the Arduino Mega microcontroller provides four serial communication ports, making the use of an additional library like “Software Serial” obsolete. Figure 16 shows the “link” on the XBee shield is removed, isolating the shield from the microcontroller’s digital I/O pins. On the shield, all the “DIN” pins are electrically connected to each other and all the “DOUT” pins are electrically connected to each over. Any “DOUT” pin of the XBee shield can, therefore, be connected to the receiving pin on communications port 1 of the microcontroller, and any “DIN” pin of the XBee shield can be connected to the transmit pin on communications port 1 of the microcontroller. Thus the XBee data communication is hard wired from the XBee shield to the communications port of the microcontroller. The same data communications approach was used for the XBee shield on the master unit.

The microcontroller decodes the MODBUS data as it is received from the serial communications port and will only reply if the address matches its own address. Every time the microprocessor detects that the SFD byte is equal to 1, it will write the decoded information to the micro-SD card via the Serial Peripheral Interface (SPI) for storage.

Figure 15 is a close-up view of the XBee shield data communications pin selector for the software serial library where the user can define the transmit-and-receive pins. It can be seen that the “links” are set for D0 and D1 using the microcontroller’s default serial communications port.

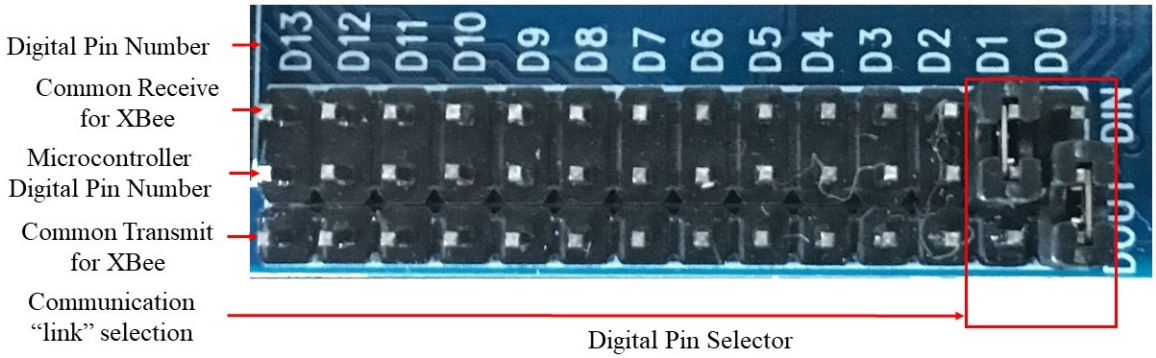


Figure 15: Digital Pin Selector on XBee Shield

3.6.1 System Components

The different shields stacked on top of each other form the XBee slave unit. They can be placed on top of each other as they are all pin compatible. Figure 18 shows the XBee radio transceiver plugged in on its own shield and the XBee transceiver shield is on top. The Arduino microcontroller is situated at the bottom, with the data logging shield sandwiched in the middle. Default I/O pins and C++ library are used for the SPI communications to the data logging shield. The microcontroller provides power to both the data logging shield and XBee transceiver shield. The different shield components and wire connections can be seen in Figure 18 below, with the full C++ code available in Annexure D. A standard C++ SD-card library is used to write the information to the SD card.

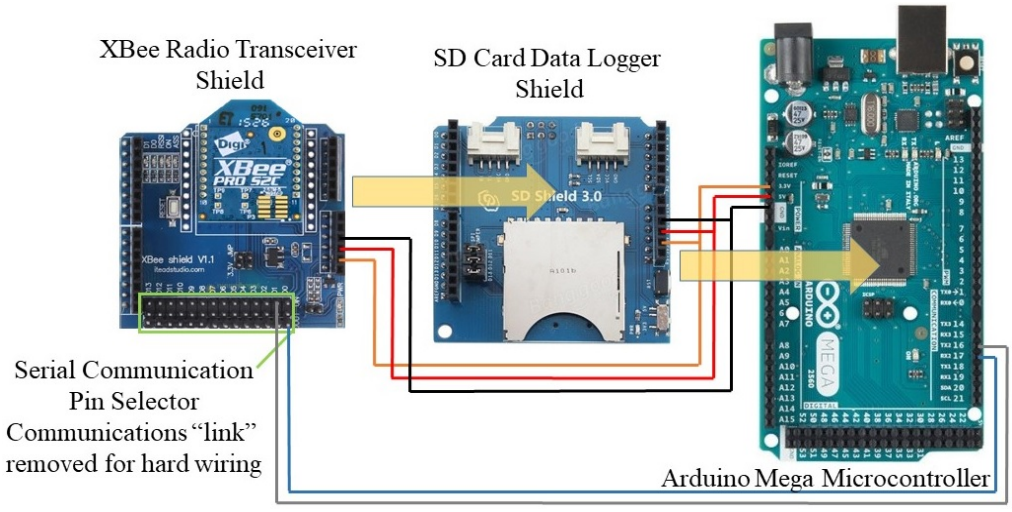


Figure 16: Sampler Unit with XBee Radio Transceiver

3.7 Wi-Fi and GPRS Class 10 Slave Units

Both the Wi-Fi and GPRS slave units are discussed in this section as their inner working is exactly the same, except that the GPRS Class 10 unit has extra Hayes AT commands to establish a data connection to the internet. The units receive data polls from the Sampler Unit via hard-wired serial cable, but will only respond to the message addressed to that specific unit. With every MODBUS "Set Multiple Registers" poll received, the units decode the information and look at the SFD byte. Normally, the value of the SFD byte is 0, but if the value of the SFD byte is 1, the microcontroller triggers the next process that will write and store the received information to ThingSpeak for cloud storage (see Annexure B, line 165, 213–224, Annexure C, line 132, 173–184).

The main units program will then reset the SFD byte value to 0 on the next poll. This will ensure that the program does not repeat the same process in the following cycle. During the decoding of the received MODBUS message, the MODBUS C++ library will populate the various registers with the newly received values (see Annexure B, line 184–211, Annexure C, line 144–171) from the master unit.

3.7.1 System Components – Wi-Fi & GPRS Class 10 Slave Unit

Both the Wi-Fi and GPRS Class 10 Unit use communications port 1 on the Arduino microcontroller to receive information from the Sampler Unit. Individually, the Wi-Fi module and GPRS Class 10 shield are connected to communications port 2 of their respective microcontrollers. As soon as the microcontroller decodes the MODBUS message and the SFD byte is set, it starts the information transfer process to the internet.

Figure 17 and 18 show the physical serial data connection between the microcontrollers to the Wi-Fi and GPRS Class 10 modules. Note that the serial MODBUS data connection between the sampler units and the slave microcontroller has been left out for simplicity's sake.

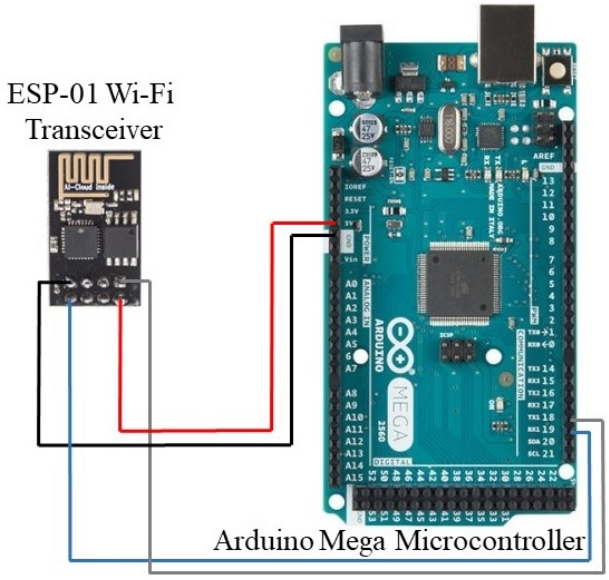


Figure 17: Arduino Mega and ESP-01 Wi-Fi Transceiver

Figure 17 also shows that the power supply of the Wi-Fi unit comes directly from the 3.3 VDC of the microcontroller, where in Figure 18, the power supply of the GPRS Class 10 comes from its own 12 VDC power supply. All external power requirements for the microcontrollers and communication devices are left out for simplicity's sake.

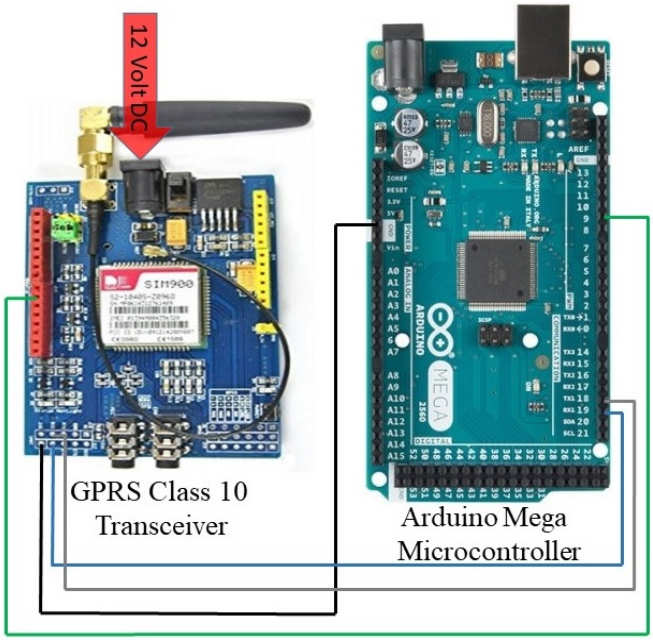


Figure 18: Arduino Mega and GPRS Class 10 Transceiver

Both the Wi-Fi and GPRS Class 10 unit use Hayes AT commands to transfer information via the internet to ThingSpeak. To achieve this, there are certain steps to follow in a sequence. The GPRS Class 10 device must first make an active connection to the internet, where the Wi-Fi unit is permanently connected to the internet via the ADSL router. Thus, there is a need for the GPRS Class 10 unit to execute an additional four AT commands to establish an active connection to the internet.

All of these AT command processes must follow in a specific order and to achieve this, a method called “State Machine” is used to control the flow from one state to another. Only if the current AT command execution process is successfully completed, will the state machine advance to the next state. If an AT command is sent to the unit, the program will wait a set amount of time for a response from the unit. If the received answer is correct, it will advance to the next state. If the answer received is wrong, the program will move back one step to the previous state and retry from there. If the answer times out with no response, the program will also move itself back one state and retry from the previous state (see Annexure B, line 291–457, Annexure C, line 251–570).

3.8 Summary

In this chapter, the practical setup of the study was discussed. This included a block diagram of the design and a detailed discussion of every unit, as well as the communications protocol used. This chapter also described the detailed working of how the master unit sets a specific byte to notify the slave units that the data to follow must be used as valid data. The remote units will then store the immediate data to the cloud service or local SD card.

The next chapter focuses on the results of the study, where data analysis is performed. A discussion of several methods used to calculate propagation delays, data integrity and external factors affecting the results are presented.

CHAPTER 4: DATA ANALYSIS AND RESULTS

4.1 Introduction

In the previous chapter, the practical setup, different components and modules used for the Sampler, XBee, Wi-Fi and GPRS Class 10 units were discussed, as well as the communications protocol. The first byte of the MODBUS payload data is called the SFD byte that is used to indicate to the slave units when they need to store the received data to the cloud. If the SFD byte value is equal to 1, the slave would start the data transfer process for data storage.

This chapter reflects the results of this study and presents empirical data. The results show the calculation of propagation delays, data integrity and cost comparisons for the practical system in this study. With the huge amount of IoT devices available and in use, factors like propagation delays, data integrity and cost analysis play a massive roll. Individuals might want to monitor only a few sensors, while large organizations might want to monitor a vast amount of sensors.

The results of this study may bring a new perspective to researchers, hobbyists and commercial developers, enabling them to make a better-informed decision about which wireless technology to choose for a given application based on their propagation delays, data integrity and affordability. In Chapter 3, the MODBUS general outline message showed that every device has a unique address. Table 11 below shows the addresses for the master and slave units used in this study.

Table 11: MODBUS Addresses

Device	Address
Sampler Unit (Master)	0
XBee (Slave 1)	1
Wi-Fi (Slave 2)	2
GPRS Class 10 (Slave 3)	3

The slave units will only respond to its own address when polled by the master unit. Although the master unit has an address of zero, it will never be addressed by any slave unit with that address value.

4.2 Propagation Delays

Propagation delay is defined in this study as the time the slave unit takes to respond to the master unit after a successful event poll. The propagation delay can be calculated using two methods. Firstly, with the aid of a protocol analyser by eavesdropping the RS232 communication line at the sampler unit, and secondly, by using the data sent to the cloud for storage in conjunction with the cloud time tag information.

4.2.1 Protocol Analyser Propagation Delay Calculation

The protocol analyser time stamps each message transmitted and received with an accuracy of 1 millisecond. By subtracting the time differences between the transmitted message and received message, the propagation poll cycle can be calculated.

4.2.1.1 Propagation Delays for Normal Polling

Table 12 below shows the polling cycle times for XBee, Wi-Fi and GPRS Class 10 under normal conditions where it is not necessary to store any data. It can be observed that the turnaround time for XBee is approximately 47 milliseconds slower than the hard-wired serial communications. This is calculated by using the Poll Cycle Response Time values in Table 12 with Equation 1 as follows:

$$XBee\ TT = Ave\ XBee - Ave\ (WiFi + GPRS) \quad (1)$$

Where: *XBee TT* = Turnaround time of XBee

Ave XBee = Average response time of XBee

Ave (WiFi + GPRS) = Average response time of Wi-Fi and GPRS

$$XBee\ TT = \left[\frac{76\ ms + 72\ ms}{2}\right] - \left[\frac{28\ ms + 26\ ms + 28\ ms + 27\ ms}{4}\right]$$

$$XBee\ TT = 74\ ms - 27\ ms$$

$$XBee\ TT = 47\ ms$$

This additional 47 milliseconds can be attributed to the additional overheads added by the XBee module to enable the propagation of data[84]. The turnaround times for Wi-Fi and GPRS Class 10 are almost equal.

Table 12: MODBUS Poll Cycle Times eavesdropped at Sampler Unit

Polled Device	Action	Message	Time	Poll Cycle Response Time (ms)
XBee	Tx	Preset Mult Regs Request Slave 1	14:24:00,401	
XBee	Rx	Preset Mult Regs Response Slave 1	14:24:00,477	76 ms
Wi-Fi	Tx	Preset Mult Regs Request Slave 2	14:24:00,499	
Wi-Fi	Rx	Preset Mult Regs Response Slave 2	14:24:00,527	28 ms
GPRS	Tx	Preset Mult Regs Request Slave 3	14:24:00,550	
GPRS	Rx	Preset Mult Regs Response Slave 3	14:24:00,576	26 ms
XBee	Tx	Preset Mult Regs Request Slave 1	14:24:00,599	
XBee	Rx	Preset Mult Regs Response Slave 1	14:24:00,671	72 ms
Wi-Fi	Tx	Preset Mult Regs Request Slave 2	14:24:00,694	
Wi-Fi	Rx	Preset Mult Regs Response Slave 2	14:24:00,722	28 ms
GPRS	Tx	Preset Mult Regs Request Slave 3	14:24:00,746	
GPRS	Rx	Preset Mult Regs Response Slave 3	14:24:00,773	27 ms

Figure 19 shows the actual MODBUS data captured with the ASE 2000 (protocol-analyser tool that can decode multiple protocols into an easy readable format) for the times used in Table 12 above. The left column of the Line Monitor view shows the hexadecimal MODBUS data to and from the master unit to the 3 slave units, while the right-hand column is the decoded and interpreted MODBUS data.

The right-hand column also shows the exact time stamp for each message with a millisecond accuracy. This time stamping information is reflected in Table 12 above, and is used to calculate the average response time for the XBee unit. The red messages represent messages from the master unit to the slave units, while the blue messages

represent the response messages from the slave units to the master unit.

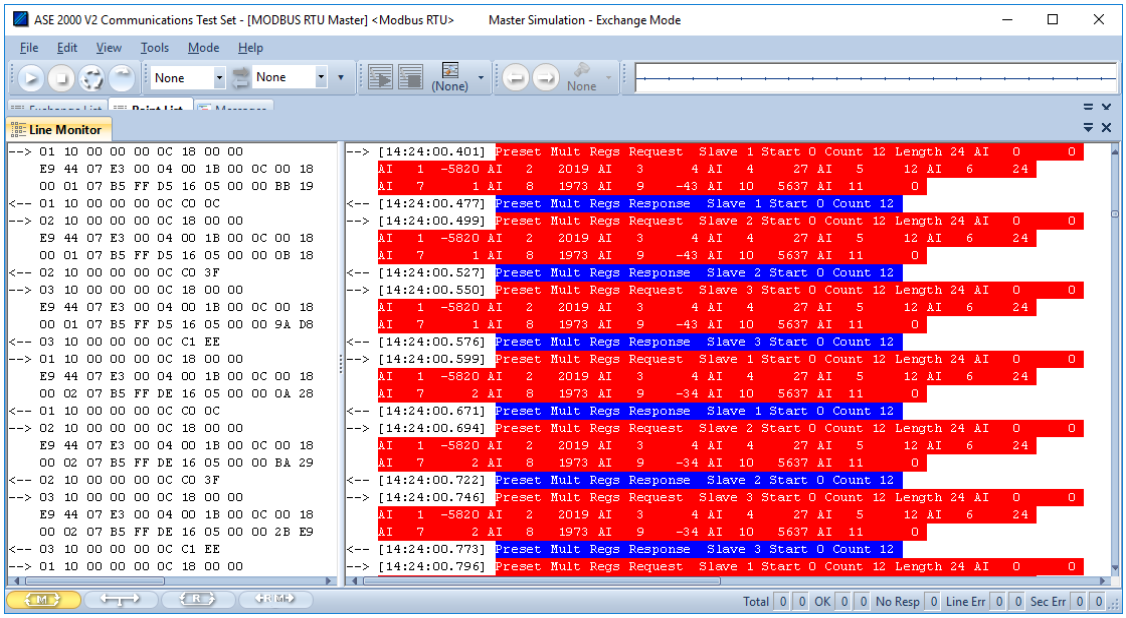


Figure 19: ASE 2000 View of Sampler Unit polling Slave Units using MODBUS

4.2.1.2 Propagation Delay for Data Storage

On average, each slave unit is updated 5 times per second (15 polls from the sampler unit to all the slave units per second). Thus, the sampler unit is polling slave units roughly every 66 milliseconds. When the SFD byte value is equal to 1, the slave units immediately start transferring the received information to the cloud via Wi-Fi, GPRS Class 10, and to the SD memory card.

If there is no reply from the slave unit, the protocol analyser will identify the null response as “Response Timeout Response”. The Arduino is a simple microcontroller-based platform without the concept of an operating system. This means that only one program can run in Arduino at a time. Therefore, if the microcontroller starts the data transfer process, it will not respond to polls received from the sampler unit. This failure to respond can then be used to calculate the time it took for the unit to send the information to the cloud for storage. Furthermore, these values can be verified on the cloud storage side, as every message is time stamped on arrival at the ThingSpeak cloud server.

4.2.1.2.1 Transmission Delay Calculated with ASE 2000 Protocol Analyser

The propagation delay for each unit can be calculated by using the transmission time stamped "Response Timeout Response" from the ASE 2000. Figure 20 and Table 13 show the time information and normal polling as seen in the ASE 2000 protocol analyser. As soon as the SFD data byte value is 1, the Arduino program starts storing the received data. Figure 20 shows a screen capture where the SFD byte value is 1 (encircled values in Figure 20), and the data in that message needs to be stored. The data for all 3 slave units is exactly the same as shown in Table 13. Table 13 is also a clearer view of the values in Figure 20.

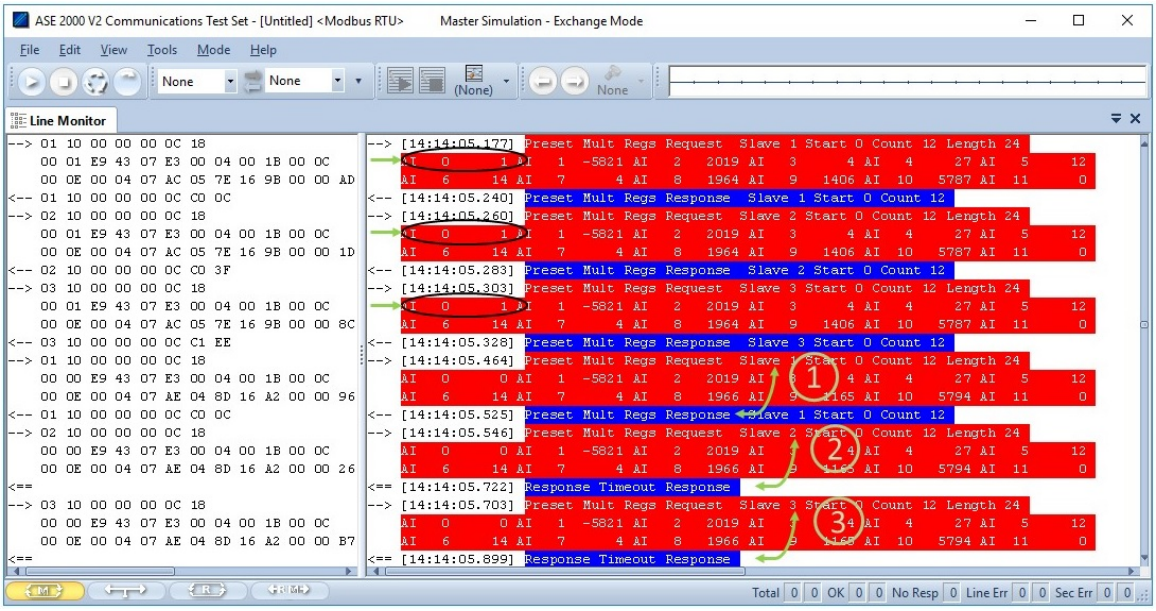


Figure 20: SFD Byte Value is set to a value of 1 and response from slave units

Note that on the subsequent polling cycle (Figure 20 encircled 1), only slave unit 1 responded to the poll from the sampler unit, meaning that slave 2 and slave 3 are still busy transmitting their data for cloud storage (Figure 20 encircled 2 and 3). Slave 1 has finished the process of writing its data to the SD card, and can respond to the sampler unit's poll.

Table 13: Values of MODBUS message to the Slave Unit

Analogue	Name	Value
A0	SFD	1
A1	Index	59715
A2	Year	2019
A3	Month	4
A4	Day	27
A5	Hour (UTC Time)	12
A6	Minute	14
A7	Second	4
A8	Voltage (Divide by 100)	1964
A9	Current (Divide by 100)	1406
A10	Temperature (Divide by 100)	5787
A11	Not Used	0

Figure 21 encircled 4 shows that the Wi-Fi Unit is still not responding to the poll from the master unit, meaning that it is still busy with the data-transfer process to cloud storage. Figure 21 encircled 5 shows that it responds to the poll from the sampler unit for the first time, meaning that the data transfer process was completed.

Figure 21 encircled 6 shows that the GPRS Class 10 Unit is still not responding to the poll from the master unit, meaning that it is still busy with the data transfer process to cloud storage. Figure 21 encircled 7 shows that it responds to the poll from the sampler unit for the first time, meaning that the data transfer process was completed. Table 14 is a combined view of all three units with their individual poll timeouts.

Figure 20 shows that the XBee Radio unit (slave unit 1) replied to the subsequent poll, meaning that within 63 milliseconds, the information was written to the SD card and the unit was ready to respond to the poll from the sampler unit. Figure 21 and Table 14 show that the Wi-Fi Unit (slave unit 2) took 2363 milliseconds to respond to a poll from the sampler unit. Figure 22 and Table 14 show that GPRS Class 10 took 5720 milliseconds to respond to a poll from the sampler unit.

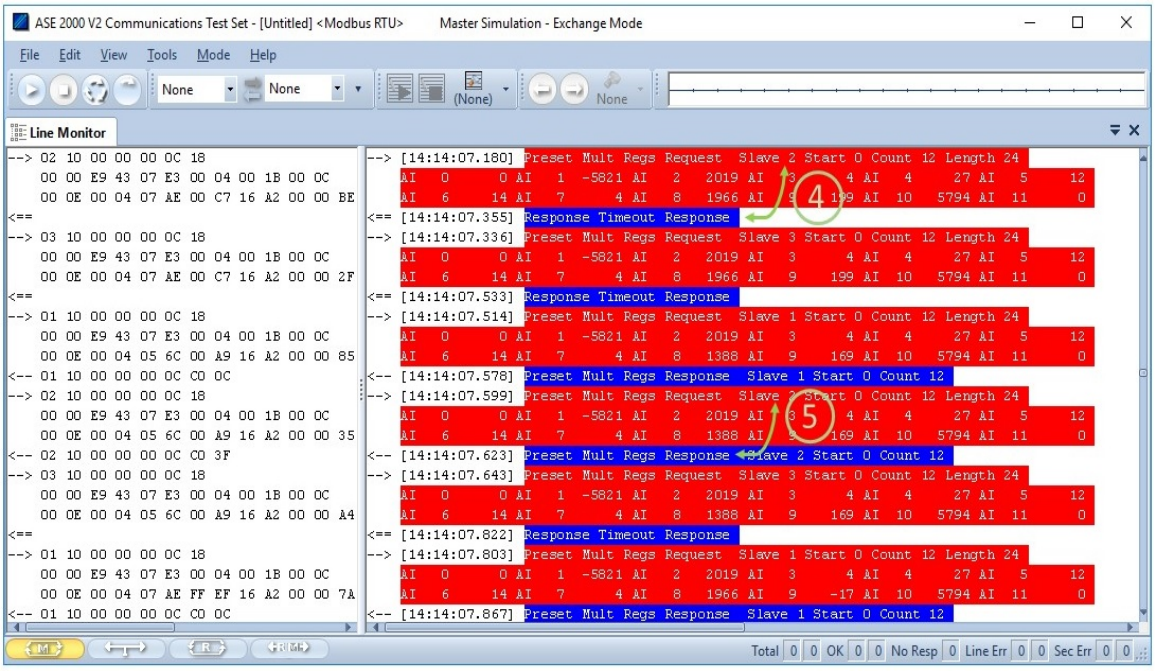


Figure 21: First Response for Wi-Fi Unit

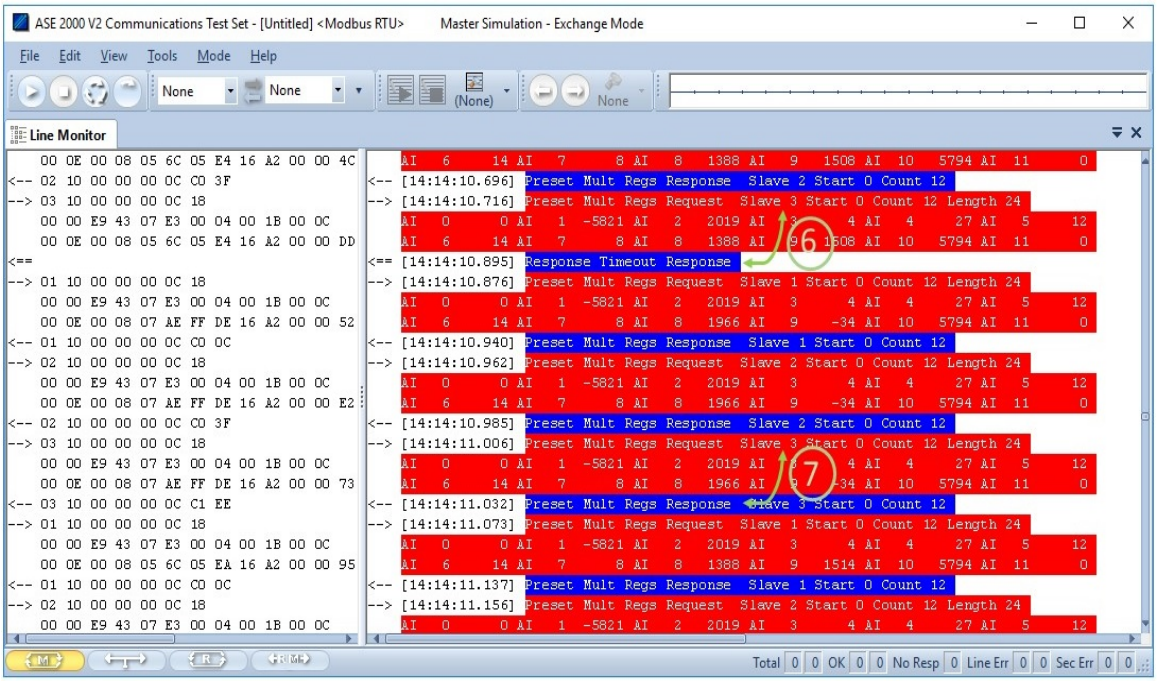


Figure 22: First Response for GPRS Class 10 Unit

Table 14: Slave Units' Response Times

Device	Action	Tx and Rx Time	First Response (ms)
XBee	Tx	14:14:05,177	
XBee	Rx	14:14:05,240	63 ms
Wi-Fi	Tx	14:14:05,260	
Wi-Fi	Rx	14:14:07,623	2363 ms
GPRS	Tx	14:14:05,303	
GPRS	Rx	14:14:11,023	5720 ms

4.2.2 Propagation Delay Calculated at ThingSpeak Cloud Server

The time in the message is the latest time received from the GPS. In this case, it was 12:14:04 UTC Time (Table 13). Hence, the propagation delay times can be verified by subtracting it from the time stamp (Table 15, TS Time = ThingSpeak Time Stamped Time) when the message arrived at ThingSpeak.

$$Time\ Delay = TS\ Time - GPS\ Time \quad (2)$$

Where: *TS Time* = ThingSpeak Time Stamped Time
GPS Time = Sampler unit GPS Time

$$WiFi\ Time\ Delay = 12{:}14{:}06 - 12{:}04{:}04$$
$$WiFi\ Time\ Delay = 2\ sec$$
and
$$GPRS\ Class\ 10\ Time\ Delay = 12{:}14{:}09 - 12{:}04{:}04$$
$$GPRS\ Class\ 10\ Time\ Delay = 5\ sec$$

Table 15 shows the time delay at the ThingSpeak cloud server. A delay of 2 seconds was recorded for the Wi-Fi data and a delay of 5 seconds was recorded for GPRS Class 10 data.

Comparing the "Time Delay" of 2 seconds with the "First Response" of 2363 milliseconds from Table 14 for Wi-Fi and the "Time Delay" of 5 seconds with the "First Response" of

5720 milliseconds from Table 14 for GPRS Class 10, it is clear that the calculated times validate 100% with the protocol analyser measured times from Table 14. Bear in mind that the protocol analyser will be more accurate as it has a 1 milliseconds resolution.

Table 15: Propagation Delay Time Using Time Stamp from ThingSpeak

Unit	Index	TS Date	TS Time	GPS Time	Temp-erature	Voltage (V)	Current (A)	Time Delay
Wi-Fi	59715	2019-04-27	12:14:06	12:14:04	57,87	19,64	1,406	00:00:02
GPRS Class 10	59715	2019-04-27	12:14:09	12:14:04	57,87	19,64	1,406	00:00:05

The response-times using different internet connections to the cloud server gave a faster response time for the Wi-Fi connection than the GPRS Class 10 connection. This is mainly due to the manner in which Wi-Fi and GPRS Class 10 connections are treated by the software when connecting to the internet. Table 16 shows that the number of Hayes AT commands to make a GPRS Class 10 connection, send the data and close the connection, is a third more than with Wi-Fi (6 AT commands for Wi-Fi, compared to 9 AT commands for GPRS Class 10).

Table 16: Hayes AT Commands for Wi-Fi and GPRS Class 10

Hayes AT Command	Wi-Fi	GPRS Class 10
Attach to GPRS CLASS 10 service	×	✓
Bring up wireless connection	×	✓
Query current connection status	×	✓
Query IP address of the given domain name	✓	✓
Start TCP connection	✓	✓
Issue Send command	✓	✓
Send data to cloud server	✓	✓
Close the connection	✓	✓
Close the connection from the cloud server	✓	✓

For each of these AT commands to transmit a packet of data to cloud storage via the internet takes time, followed by a timeout period to ensure successful data transmission for both

Wi-Fi and GPRS Class 10. The Hayes AT command comparison from Table 16 for Wi-Fi and GPRS Class 10 was also presented at the SAUPEC 2019 conference in 2019 [85] (See Annexure F).

Figure 23 shows a graphical view of the transmission delays for both Wi-Fi (red graph) and GPRS Class 10 (blue graph) over a period of 6 months. From the graph, it is noticeable that the maximum transmission time for Wi-Fi was 4 minutes, 27 seconds, and for GPRS Class 10 it was 4 minutes, 14 seconds. This phenomenon occurred only once for Wi-Fi and GPRS Class 10 over the evaluation period. It is also visible from the graph that the Wi-Fi unit took 17 times more than 1 minute and 26 seconds to transmit its data, compared to the 7 times for GPRS Class 10. Figure 23 was presented at the SATNAC 2019 conference in 2019 [86] (See Annexure G).

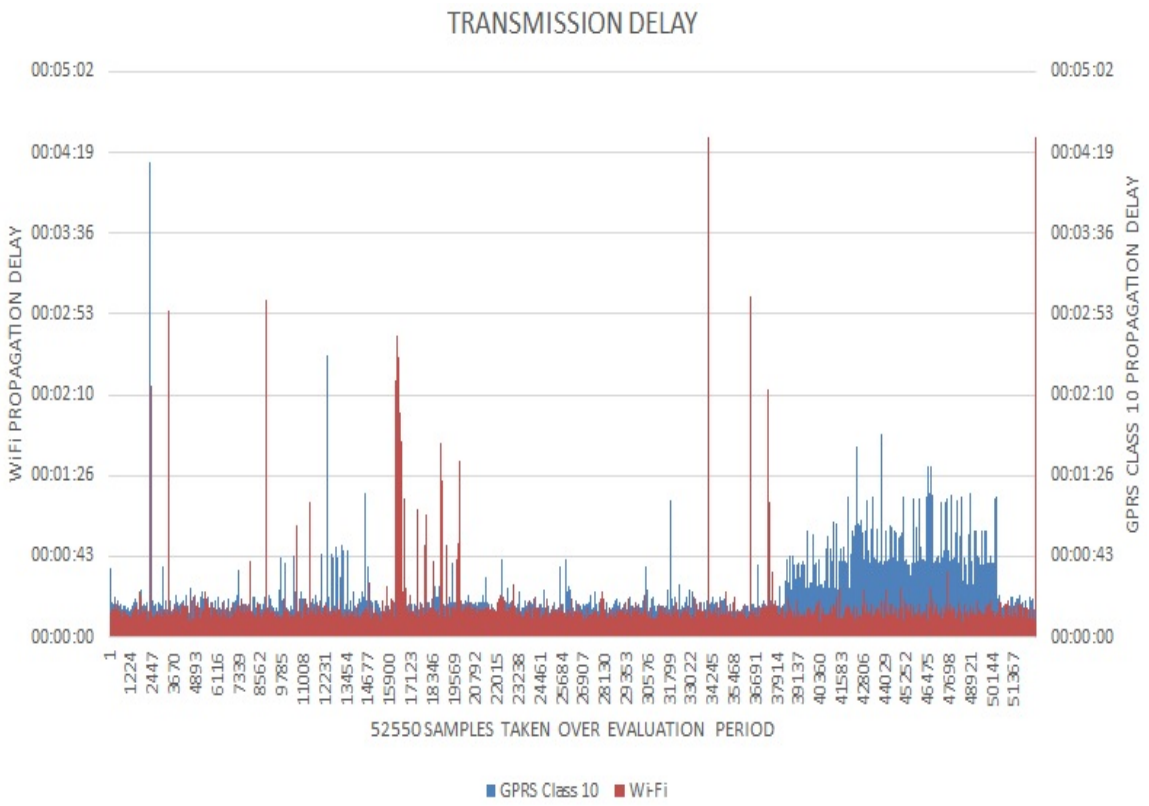


Figure 23: Data Transmit Delay Times to ThingSpeak

(52550 samples taken over 209 days every 5 minutes)

4.3 Data Integrity

Information is not transmitted across a network as a single continuous stream, but rather as a series of discrete units, called packets. These packets are like individual pages in a book. Individually, they do not mean much. Together, they have some meaning, but only when they are connected to each other in the correct order. When a network connection drops a packet of data, the full book cannot be constructed. Packets can also arrive incomplete, damaged or flawed, causing them to be useless. To fix this, a typical resend strategy is needed. In this study, the resend strategy was deliberately left out by ignoring the confirmation of an incomplete package response from the cloud storage service, so that the reliability of XBee radio, Wi-Fi and GPRS Class 10 could be determined. Significant packet loss is like a badly leaking pipe.

Data packet loss is defined as a ratio of the number of packets sent by the master unit to the number of successful packets received by the cloud storage application [87]. Data packet loss occurs when a network connection loses information while in transit and may be correlated to data integrity. It can make a network connection seem slower than it should be and reduces the reliability of network communication with both local and remote devices.

A total of 52 550 data packets were sent over the 209-day period. Figure 24 shows the total amount of data packets that were lost for this period; 123 data packets for XBee radio; 185 data packets for Wi-Fi; and 872 data packets for GPRS Class 10. Calculating the reliability based on these figures indicates that the XBee radio system is much more reliable with 0.21% data packets lost, closely followed by Wi-Fi with 0.31% and GPRS CLASS 10 with 1.46%.

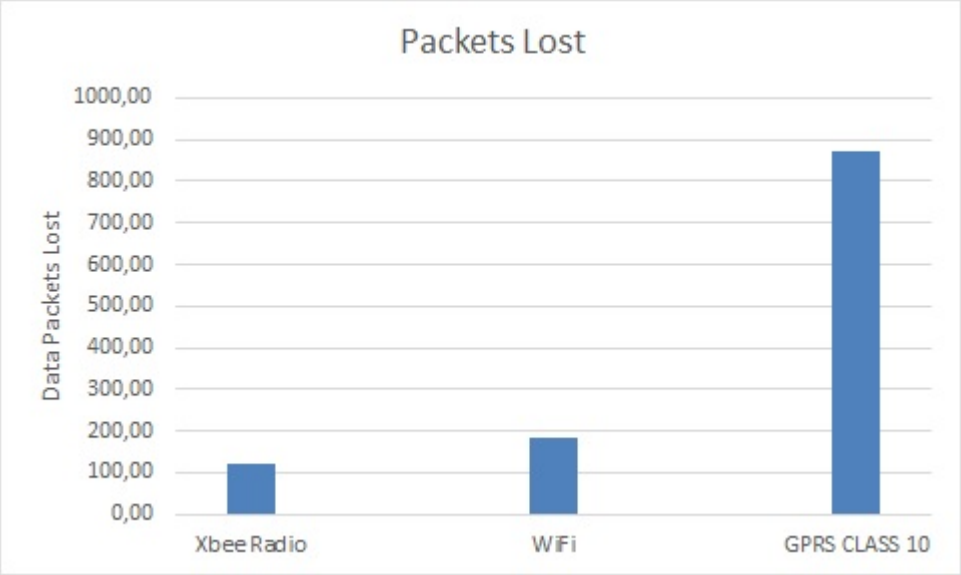


Figure 24: Total Data Packets Lost

It is evident from Figure 24 that packet loss does occur for all three technologies under test. However, data loss incidents are relatively rare and hard to quantify. Other studies showed that the expected packet loss over XBee is 0 % for distances less than 50 m [88], 0.5 % for Wi-Fi [89] and GPRS Class 10 is about 0.83% [90].

4.3.1 Data Errors

Data errors are defined as a condition in which digital data gets altered or corrupted within a data packet [91]. For example, “Receivers of voice and video streams are very tolerant of transient errors, which shows up as pops in your ear or static on a screen. Data is more sensitive and usually handled with retransmission requests” [91]. Data errors are possible with the transfer of data to the internet but fortunately, TCP/IP makes use of error detection and error correction on the second layer of the OSI model.

TCP/IP uses error correction and data stream control techniques to ensure that the packet arrives at its intended destination uncorrupted and in the correct sequence, making the point-to-point connection virtually error-free [4]. Out of the 52 550 packets transferred to ThingSpeak, there were 0 errors to report for both Wi-Fi and GPRS Class 10 technologies.

4.4 Load Shedding

The loss of data is not always a result of data errors, but can also be effected by external factors like load shedding. Uninterruptable power sources are costly and not always available for all the communication equipment, like ADSL routers that depend on the mains supply from the local energy provider. Any failure or interruption of the mains supply will affect the transfer of data.

Since the 1920s, power-system stability is known to require a stable energy source. This phenomenon has been illustrated by many major blackouts caused by power system instability. In highly stressed conditions, different forms of system instability have emerged. For example, voltage stability, frequency stability and inter-area oscillations have become of great concern. The IEEE/CIGRE task force defined the idea of energy source stability in the following way: "Power system stability is the ability of an electric power system, for a given initial operating condition, to regain a state of operating equilibrium after being subjected to a physical disturbance, with most system variables bounded so that practically the entire system remains intact" [92].

A last resort to prevent the collapse of the power system is to implement load shedding. The electricity system becomes unbalanced when there is an insufficient supply of power from the power stations and the demand (load) becomes too high, which can cause the network to trip countrywide. This could take days to restore. The power utility can thus either increase supply or reduce demand to bring the system back into balance. As the difference between supply and demand becomes small, we refer to the system becoming "tight". This implies that action must be taken to prevent system instability.

There is usually a sequence of steps that need to be taken to avoid load shedding. First, the power utility would ask large customers to reduce their load voluntarily. However, if several power units trip unexpectedly or the voluntary reduction in load is not enough, the power utility may have to implement load shedding to prevent the system from becoming unstable.

Scheduled load shedding is controlled by sharing the available electricity among all its customers. By switching off parts of the network in a planned and controlled manner, the system remains stable throughout the day, and the impact is spread over a wider base of customers.

Figure 25 shows the availability of the Wi-Fi router and the power failure interruptions due to load shedding for the period of 1 to 6 December 2018. It is perceived from Figure 25 that there was a normal stage 2 load shedding for the first 3 days where the electrical energy supply was interrupted once a day. Then on day 4 (encircled Day 4 in Figure 25), stage 4 load shedding was implemented and the supply was interrupted twice from 07:04 to 08:25 in the morning and again from 18:15 to 20:01 in the evening.

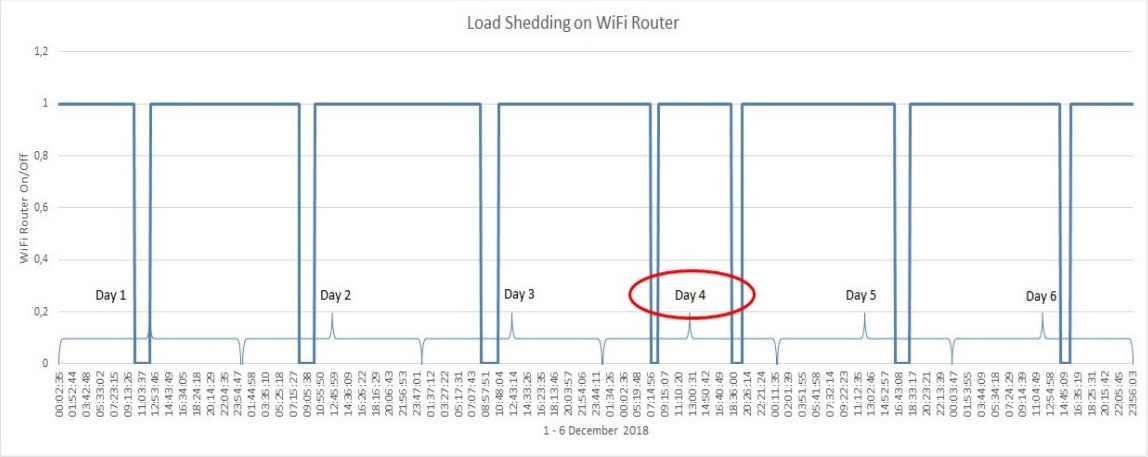


Figure 25: Wi-Fi Router Supply Loss Due To Load Shedding

4.5 Cost Analysis

Although it is important to look at propagation delay and data integrity, the affordability to transfer data to a cloud service is also a concern. Table 17 shows the cost comparison of all three technologies used in the study. Note that the cost is split into two categories, namely "Once-Off Cost" and "Per-Month Rate". It can be seen from the table that the XBee unit has an initial cost of R823.00 with no monthly charges, nor does it have internet connectivity. For it to have internet connectivity, Wi-Fi or GPRS Class 10 must be added.

The relatively low initial cost for the Wi-Fi unit is R223.00, but the monthly recurring cost is the highest of them all at R450.00. However, it gives the user an unlimited amount of data to transmit or receive information.

The second most expensive unit is the GPRS Class 10 unit at R663.00, due to the complexity of the infrastructure and technology used. With fairly low monthly costs of R100.00, it gives a limited amount of data. Once the data bundle is depleted, the connection to the internet is also suspended.

If the amount of data that needs to be sent to the internet is fairly high, it might be more cost effective to use a Wi-Fi system with an ADSL or fibre connection and an unlimited amount of data compared to GPRS Class 10. This type of internet connection cannot always be achieved as the sensor that needs to be monitored could be in a very remote location, making GPRS Class 10 more cost effective.

Table 17: XBee, Wi-Fi and GPRS Class 10 Cost Comparison

	Once Off Cost			Per Month Rate		
	Hardware	Additional Shields	**Total**	Line Rental	ISP/Data	**Total**
XBee	R698,00	R125,00	**R823,00**	R0,00	R0,00	**R0,00**
Wi-Fi	R68,00	R155,00	**R223,00**	R300,00	R150,00	**R450,00**
GPRS Class 10	R663,00	R0,00	**R663,00**	R0,00	R100,00	**R100,00**

4.6 Summary

This chapter focused mainly on the analysis and comparison of the data sets for XBee radio, Wi-Fi and GPRS Class 10, to determine propagation delay, data integrity and to do a cost analysis of all three technologies under test. XBee proved to be the most reliable with the least amount of data loss, but comes with the disadvantage that it does not have a real-time connection to the internet.

Wi-Fi might be the most expensive on a monthly basis, yet it seems to be a very reliable

data-communication bearer. The higher cost can be quantified if a large number of sensors need to be connected to the internet for data storage. Although the initial cost of GPRS Class 10 might be the most expensive, it is the cheapest method to connect remote sensors to the internet for cloud storage of sampled data.

The analysis of the data showed that the reliability and data loss over all three technologies evaluated is remarkably similar and acceptable to use from any platform. Nevertheless, the analysis also showed that external factors like load shedding can have a major effect on the amount of data packets that lost if you do not have an adequate power backup system. The following chapter discusses the overall outcome of the research and presents the interpretation of the results.

CHAPTER 5: DISCUSSIONS, IMPLICATIONS AND CONCLUSION

5.1 Introduction

The previous chapter covered the data analysis and the results of the study. the propagation delay, data integrity and cost analysis of the study were mainly done by analysis, calculation and comparison of the various data sources from each technology, forming a quantitative study. In this chapter, a reflection of the previous chapters will be done, the research question and objectives will be addressed, and final recommendations will be given.

5.2 Reflection of the previous Chapters

Chapter 1: The background presented individual IoT devices that can connect to the internet and transfer data to a cloud service like ThingSpeak. The use of cloud services, like ThingSpeak, enables researchers and hobbyists to access accumulated data from any platform, anywhere in the world, at any time, with the added advantage that their data is always backed up. However, this makes it challenging to determine the most cost-effective and reliable way to transfer this sampled data from a strategic location to an IoT server, like ThingSpeak. Three possible solutions were investigated and evaluated, namely XBee Digital Radio, Wi-Fi and GPRS Class 10.

The three research questions that were posed are answered, as well as the four objectives stated, are addressed in the following sections of Chapter 5. The brief methodology section incorporated a basic block diagram of the system, where the main results of the study would focus on propagation delay, data integrity and percentage of data packets lost. The benefits of the study will give researchers, students and hobbyists a better understanding of the pros and cons of the different wireless technologies offered. The delimitations of the study were also covered, which included focusing on only three specified wireless technologies.

Chapter 2: The literature study was done on relevant available work and literature. A suitable data source was needed to generate data for the study. Solar energy, wind power and biomass were considered as possible data sources. It was decided to use a solar energy system, due to the availability of the components and simplicity of the system. A pico-solar system is also economically viable and compact, requiring a much smaller installation.

MODBUS was discussed as an industry-standard protocol used by most PLCs and SCADA equipment and was chosen as the communications protocol for this research to transfer the measured data from the pico-solar system to the chosen wireless communication technologies. A communication protocol is a set of rules that must be obeyed by all users, and it specifies a common format so that all nodes know how to parse and construct data packets to and from each other.

A review of XBee modules revealed that they are embedded solutions providing wireless end-point connectivity to devices. They are easy to set up, designed for high-throughput applications requiring low latency and predictable communication timing. XBee modules provide a simple RS232 communication interface for easy integration with many projects.

A review of Wi-Fi revealed that this technology uses radio waves to provide network connectivity without requiring a physical wire connection. A Wi-Fi connection is established using a wireless adapter to create hotspots in the vicinity of a wireless router that is connected to the internet.

A review of General Packet Radio Services (GPRS) revealed that it was the first evolutionary step in deploying a truly mobile packet-based wireless communication service. It promises a direct internet connection for mobile phones, microprocessors and computers. It can provide idealised data rates between 56 and 114 kB/s.

Although there are several cloud service providers currently emerging into the market who support a large number of IoT-based devices, it was decided to focus only on

ThingSpeak. This is mainly due to its simplicity, available examples and its ability to let the user visualize and analyse live data streams in the cloud.

Chapter 3: In this chapter, the practical setup of the study was discussed. The three wireless technology systems and sampling unit were designed and built with standard off-the-shelf electronic equipment. Each unit is based on an Arduino Mega 2560 microcontroller along with each unit's particular shields and supporting components. Cyclic measurements of the output voltage, current and temperature of a PV panel was taken every 5 minutes. The sampling unit also recorded the sampled values to an on-board data logger track the sampled values using an indexing system.

The XBee radio unit used an Arduino Mega 2560 microcontroller with an XBee shield on top, as well as an SD card data-logging shield for data storage. The two XBee radios were set up to communicate with each other in the same network. When data had to be stored, the Arduino microcontroller would access the SD card and append the newest information to the log file.

The Wi-Fi Transmitter unit used an Arduino Mega 2560 microcontroller with an external ESP-01 Wi-Fi module. At start-up, the Arduino microcontroller connected and authenticated itself to the internet-connected Wi-Fi router. When data had to be sent for cloud storage, the Arduino microcontroller will create a connection to ThingSpeak via the Wi-Fi module.

The GPRS Class 10 Transmitter unit used an Arduino Mega 2560 microcontroller with a serially-connected GPRS CLASS 10 module. When data had to be sent for cloud storage, the Arduino microcontroller would establish an internet connection via the GPRS CLASS 10 module to transfer the data to ThingSpeak.

Chapter 4: Data analysis and results were discussed in this chapter. Propagation delay was defined as the time the slave unit takes to respond to the master unit after a successful event poll. The propagation delay can be calculated using two methods, namely with the aid of a

protocol analyser, or by using the time tag information.

Data packet loss was defined as the ratio of the number of packets sent by the master unit to the number of successful packets received by the cloud storage application. Data packet loss occurs when a network connection loses information while it is in transit.

Data errors were defined as a condition in which digital data gets altered or corrupted within a data packet. Data errors are possible with the transfer of data to the internet, but fortunately, TCP/IP makes use of error detection and error correction on the second layer of the OSI model.

The loss of data could also be a result of external factors like load shedding. Data transfer will be intermittent when equipment without a backup power supply fail, when the mains supply is interrupted.

Along with propagation delay and data integrity, the affordability to transfer data to a cloud service is also a concern. The XBee unit only has an initial cost with no monthly charges, as it does not require constant connectivity to the internet. The Wi-Fi unit has a relatively low initial cost, but the monthly recurring cost is the highest of all three technologies. The GPRS Class 10 unit is the most expensive due to its complex infrastructure and technology. The monthly cost is low because the amount of data is limited.

5.3 Research Questions

The research questions, as stated in Chapter 1, can now be answered as follows:

- What technologies are available to transfer data via the internet to a cloud server for storage?

Only three of the huge number of data communication technologies were evaluated.

Other wireless air interfaces linked to IoT include Bluetooth, LoRa WAN, Sigfox, ultra-wideband, near-field communication, radio-frequency identification, and more. The wireless communication technologies that were evaluated in this research included XBee, Wi-Fi and GPRS Class 10. Ultimately, the combination of data communication technologies available could make the transmission of sensor data very reliable and cost-effective.

- Which technology is more reliable?

The calculated reliability based on the 209-day evaluation period indicates that the XBee radio system is the most reliable system with 0.21% data packets lost, closely followed by Wi-Fi with 0.31% and GPRS CLASS 10 with 1.46%. From this, it can be accepted that packet loss does occur for all three technologies under test. However, data loss events are relatively intermittent and hard to account for.

- Are there significant time delays between the technologies?

From Chapter 4, it is notable that GPRS Class 10 is on average 4 seconds slower than Wi-Fi for the transmission of data to cloud storage. Table 18 shows that the radio connection was less than 0.1 second. The maximum time it took for the GPRS Class 10 to transmit a packet of data to the cloud server was 4 minutes and 14 seconds (254 sec), compared to the maximum time of 4 minutes and 27 seconds (267 sec) for Wi-Fi. The minimum time for the GPRS Class 10 was 5 seconds, compared to 2 seconds for Wi-Fi.

XBee digital radio communication has the advantage over GPRS Class 10 and Wi-Fi communication because it does not have any internet communication overheads to adhere to, and can send its data without delay. The drawback, however, is that the XBee digital radio needs a third-party communication device to connect to the internet.

Table 18: Average, Max and Min Transmit Times for Radio, Wi-Fi and GPRS Class 10 in Seconds.

	XBee Radio	Wi-Fi	GPRS CLASS 10
Average	>0.1sec	4 sec	8 sec
Maximum	>0.1 sec	267 sec	254 sec
Minimum	>0.1 sec	2 sec	5 sec

- Which technology is more cost-effective?

The initial setup cost of Wi-Fi is definitively a cost-effective way to connect IoT devices to the internet for data communications, but the downside is that the geographical footprint is not very large. Normally, every microprocessor needs a Wi-Fi module and it costs extra to connect these along with the router to the internet, plus a monthly subscription to an internet service provider.

If a large number of IoT sensors were monitored with a small geographical footprint, Wi-Fi will be a very cost-effective choice. If the geographical footprint is large with a huge amount of sensors, the use of XBee radios working to a centralised data concentrator will be the most cost-effective, combined with Wi-Fi or a GPRS Class 10 communication module for internet connectivity. If only one or two sensors are monitored in a remote destination, the use of GPRS Class 10 will be the most cost-effective.

5.4 Objectives

In Chapter 1, the following objectives were stated that are now addressed:

- Formulate a system design and determine the overall installation and maintenance costs for each technology;

The three technologies that were evaluated are XBee digital radio, Wi-Fi and GPRS Class 10. A complete working model was built to determine the individual complexity, installation cost and maintenance cost for each. From a hardware point of view, XBee was the most expensive with GPRS Class 10 second and Wi-Fi the least expensive. The monthly

maintenance cost for XBee was the lowest with GPRS Class 10 secondly and Wi-Fi the most expensive. Bear in mind that XBee digital radios do not have a connection to the internet, so internet connectivity would come at additional cost.

- Analyse the simplicity of each technology;

The simplest installation was the XBee unit. The available XBee shield make it a plug-and-play exercise, while the versatility and simplicity of the XBee shield made it very easy to connect directly to the microprocessor's communication port. Data communications with the XBee digital radios were fairly easy, as the XBee module handles all the data communication handshaking. Secondly was the Wi-Fi unit with an external Wi-Fi module. It also connects serially with the microprocessor, but needed some special Hayes commands to establish a connection with the Wi-Fi router and to send data to ThingSpeak. The GPRS Class 10 unit was the most complex. It also connects serially to the microprocessor, but requires additional Hayes commands to register itself to the GSM network, establish a connection and send data to ThingSpeak.

- Determine the ease of implementation and power consumption for each technology;

All three technologies were fairly easy to implement as they all connected serially to the microprocessors and could function as an individual add-on. The power consumption for the XBee unit is 120 mA in transmit mode, 31 mA in receive mode and 1 μA in standby mode. The Wi-Fi unit's power consumption is 140 mA in transmit mode, 56 mA in receive mode and 15 mA in standby mode. On the other hand, the GPRS Class 10 unit is a bit more power-hungry, with 453 mA in transmit and receive mode and 18 mA in standby mode.

- Assess the data integrity for each technology.

When a packet of data gets lost over a computer network, it can be acknowledged that the packet of data that travelled across the network failed to reach its destination. This could be caused by errors in data transmission or classically over wireless networks. It can be

measured as a percentage of packets lost with reverence to packets sent. It is relatively rare to lose data and hard to identify exactly where it got lost. GPRS Class 10 was the worst with 1.46%, Wi-Fi with 0.31% and XBee radio with 0.21%.

5.5 Recommendations

Data communications worldwide is a vast growing environment with limitless boundaries. This fast-paced growth makes it very difficult to evaluate every possible IoT data communications scenario. With this in mind, here are some recommendations for future studies that may affect the outcome of this study:

- Evaluate other online cloud services;
- Evaluate various mobile service providers;
- Compare ADSL to Fibre; and
- Evaluate 5G.

References

[1] M. Rouse and I. Wigmore, “What is Internet of Things (IoT)? - Definition from WhatIs.com,” *IoT Agenda*, 2016. [Online]. Available: http://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT. [Accessed: 11-Oct-2017].

[2] C. Wang, S. S. M. Chow, Q. Wang, K. Ren, and W. Lou, “Privacy-preserving public auditing for secure cloud storage,” *IEEE Transactions on Computers*, vol. 62, no. 2. pp. 362–375, Feb-2013, doi: 10.1109/TC.2011.245.

[3] D. Y. Fong, “Wireless sensor networks,” in *Internet of Things and Data Analytics Handbook*, Wiley, 2017, pp. 197–213.

[4] W. Shang, Y. Yu, L. Zhang, and R. Droms, “Challenges in IoT Networking via TCP/IP Architecture,” *NDN Project, Tech. Rep. NDN-0038*, vol. 8, no. 2. International Journal of Science and Research, p. 7, 2016.

[5] L. Stougie, N. Giustozzi, H. van der Kooi, and A. Stoppato, “Environmental, economic and exergetic sustainability assessment of power generation from fossil and renewable energy sources,” *International Journal of Energy Research*, vol. 42, no. 9. John Wiley and Sons Ltd, pp. 2916–2926, 01-Jul-2018, doi: 10.1002/er.4037.

[6] S. Talari, M. Shafie-khah, G. J. Osório, J. Aghaei, and J. P. S. Catalão, “Stochastic modelling of renewable energy sources from operators’ point-of-view: A survey,” *Renewable and Sustainable Energy Reviews*, vol. 81. pp. 1953–1965, 2018, doi: 10.1016/j.rser.2017.06.006.

[7] J. J. Cheng, *Biomass to Renewable Energy Processes*, 2nd Ed. North Carolina: CRC Press, 2017.

[8] T. Khatib, I. A. Ibrahim, and A. Mohamed, “A review on sizing methodologies of photovoltaic array and storage battery in a standalone photovoltaic system,” *Energy Conversion and Management*, vol. 120. pp. 430–448, 15-Jul-2016, doi: 10.1016/j.enconman.2016.05.011.

[9] “Solar Power Lights up De Grendel – De Grendel Wines.” [Online]. Available: https://degrendel.co.za/blogs/news/solar-power-lights-up-de-grendel. [Accessed: 14-Sep-2020].

[10] E. Kabir, P. Kumar, S. Kumar, A. A. Adelodun, and K. H. Kim, “Solar energy: Potential and future prospects,” *Renewable and Sustainable Energy Reviews*, vol. 82. pp. 894–900, 2018, doi: 10.1016/j.rser.2017.09.094.

[11] M. Rodrigues *et al.*, “Elaboration of a Solar Stove and Analysis of Its Efficiency with a Lens.”

[12] P. G. V. Sampaio and M. O. A. González, “Photovoltaic solar energy: Conceptual framework,” *Renewable and Sustainable Energy Reviews*, vol. 74. pp. 590–601, 2017, doi: 10.1016/j.rser.2017.02.081.

[13] T. M. Razykov, C. S. Ferekides, D. Morel, E. Stefanakos, H. S. Ullal, and H. M. Upadhyaya, “Solar photovoltaic electricity: Current status and future prospects,” *Solar Energy*, vol. 85, no. 8. pp. 1580–1608, Aug-2011, doi: 10.1016/j.solener.2010.12.002.

[14] B. Parida, S. Iniyan, and R. Goic, “A review of solar photovoltaic technologies,” *Renewable and Sustainable Energy Reviews*, vol. 15, no. 3. pp. 1625–1636, Apr-

2011, doi: 10.1016/j.rser.2010.11.032.
[15] A. Jäger-Waldau, “European Photovoltaics in world wide comparison,” *Journal of Non-Crystalline Solids*, vol. 352, no. 9-20 SPEC. ISS. pp. 1922–1927, Jun-2006, doi: 10.1016/j.jnoncrysol.2005.10.074.
[16] F. Blaabjerg and K. Ma, “Wind Energy Systems,” *Proceedings of the IEEE*, vol. 105, no. 11. Springer International Publishing, Cham, pp. 2116–2131, 2017, doi: 10.1109/JPROC.2017.2695485.
[17] H. Allamehzadeh, “Wind energy history, technology and control,” *2016 IEEE Conference on Technologies for Sustainability, SusTech 2016*. Institute of Electrical and Electronics Engineers Inc., Phoenix, AZ USA, pp. 119–126, Apr-2017, doi: 10.1109/SusTech.2016.7897153.
[18] W. Shepherd and L. Zhang, *Electricity generation using wind power*, 2nd Ed. Bradford, England: WORLD SCIENTIFIC, 2011.
[19] R. McKenna, P. Ostman, and W. Fichtner, “Key challenges and prospects for large wind turbines,” *Renewable and Sustainable Energy Reviews*, vol. 53. Elsevier Ltd, pp. 1212–1221, Jan-2016, doi: 10.1016/j.rser.2015.09.080.
[20] World Wind Energy Organization, “Wind Power Capacity Reaches 539 GW, 52,6 GW Added in 2017,” *Press Release*, 2018. [Online]. Available: http://wwindea.org/blog/2018/02/12/2017-statistics/. [Accessed: 30-Oct-2019].
[21] Acciona, “Gouda Wind Farm.” [Online]. Available: https://www.acciona-energia.com/areas-of-activity/wind-power/major-projects/gouda-wind-farm/. [Accessed: 15-Nov-2019].
[22] C. Wyman, A. Wiselogel, S. Tyson, and D. Johnson, “Biomass Feedstock Resources and Composition,” in *Handbook on Bioethanol*, Routledge, 2018, pp. 105–118.
[23] Z. Al-Hamamre, M. Saidan, M. Hararah, K. Rawajfeh, H. E. Alkhasawneh, and M. Al-Shannag, “Wastes and biomass materials as sustainable-renewable energy resources for Jordan,” *Renewable and Sustainable Energy Reviews*, vol. 67. pp. 295–314, Jan-2017, doi: 10.1016/j.rser.2016.09.035.
[24] P. McKendry, “Energy production from biomass (part 1): Overview of biomass,” *Bioresource Technology*, vol. 83, no. 1. pp. 37–46, 2002, doi: 10.1016/S0960-8524(01)00118-3.
[25] P. McKendry, “Energy production from biomass (part 2): Conversion technologies,” *Bioresource Technology*, vol. 83, no. 1. pp. 47–54, 2002, doi: 10.1016/S0960-8524(01)00119-5.
[26] V. K. W. S. Araujo, S. Hamacher, and L. F. Scavarda, “Economic assessment of biodiesel production from waste frying oils,” *Bioresource Technology*, vol. 101, no. 12. pp. 4415–4422, 2010, doi: 10.1016/j.biortech.2010.01.101.
[27] K. Kaygusuz and M. F. Türker, “Biomass energy potential in Turkey,” *Renewable Energy*, vol. 26, no. 4. pp. 661–678, 2002, doi: 10.1016/S0960-1481(01)00154-9.
[28] R. Spalding-Fecher, H. Winkler, and S. Mwakasonda, “Energy and the World Summit on Sustainable Development: What next?,” *Energy Policy*, vol. 33, no. 1. pp. 99–112, Jan-2005, doi: 10.1016/S0301-4215(03)00203-9.
[29] K. Appunn, “Bioenergy - the troubled pillar of the Energiewende,” *Clean Energy Wire*, 2016. [Online]. Available: https://www.cleanenergywire.org/dossiers/bioenergy-germany. [Accessed: 15-Nov-2019].

[30] C. Morris, “Biomass – the growth is over in Germany – Energy Transition,” 2015. [Online]. Available: https://energytransition.org/2015/07/biomass-growth-is-over/. [Accessed: 14-Sep-2020].
[31] P. E. Hertzog and A. J. Swart, “Detecting the presence of pigeons on PV modules in a pico-solar system,” *2017 IEEE AFRICON: Science, Technology and Innovation for Africa*. Cape Town, South Africa, pp. 1528–1533, 2017, doi: 10.1109/AFRCON.2017.8095709.
[32] A. Delai, A. Miyadaira, and T. Lima, “AMASP (ASCII Master Slave Protocol): a Lightweight MODBUS Based Customizable Communication Protocol for General Applications,” *Journal of Communication and Information Systems*, vol. 34, no. 1. Sociedad Brasileira de Telecomunicacoes, pp. 1–11, 2019, doi: 10.14209/jcis.2019.1.
[33] A. N. Gaidhane and M. P. Khorgade, “FPGA implementation of serial peripheral interface of Flex-Ray controller,” *Proceedings - 2011 UKSim 13th International Conference on Modelling and Simulation, UKSim 2011*. pp. 128–132, 2011, doi: 10.1109/UKSIM.2011.33.
[34] J. D. Senra Simoes, E. A. de Seabra, and A. M. Fernandes, “Study, Design and Development of a Modbus Master That Evaluates the Modbus Communication Between Equipments,” *Proceedings of the 7th International Conference on Mechanics and Materials in Design (M2D2017)*. INEGI-FEUP, Albufeira, Portugal, pp. 1817–1818, 2017.
[35] L. E. Frenzel, *Handbook of serial communications interfaces : a comprehensive compendium of serial digital input/output (I/O) standards*. 2015.
[36] I. N. Fovino, A. Carcano, M. Masera, and A. Trom-Betta, “Design and implementation of a secure Modbus protocol,” *IFIP Advances in Information and Communication Technology*, vol. 311. Springer New York LLC, pp. 83–96, 2009, doi: 10.1007/978-3-642-04798-5_6.
[37] J. Makhija, “Comparison of Protocols Used in Remote Monitoring: DNP 3.0, IEC 870-5-101 & Modbus,” *Electronics Systems Group*, no. 03307905. pp. 1–19, 2003.
[38] F. Yncio, R. Peña, A. Cadiboni, R. Fernández, G. Ahrtz, and C. Sosa Tellechea, “A Modbus client for the identification of an energy recovery system for a water distribution network,” *2018 IEEE 9th Power, Instrumentation and Measurement Meeting, EPIM 2018*. Salto, Uruguay, pp. 1–6, 2018, doi: 10.1109/EPIM.2018.8756337.
[39] N. Dutt, Y. Mathur, A. Pandey, and D. Mathuria, “Application of MODBUS and RS485 technologies to accelerator controls at IUAC,” *Application of MODBUS and RS485 technologies to accelerator controls at IUAC*. New Delhi, India, p. 352, 2015.
[40] M. Jagadesh, M. Saravanan, V. Narayanan, M. Priya Vadhana, and K. Logeshwaran, “Monitoring System in Industry Using IoT,” *2019 5th International Conference on Advanced Computing and Communication Systems, ICACCS 2019*. Coimbatore, Tamil Nadu, pp. 745–748, 2019, doi: 10.1109/ICACCS.2019.8728324.
[41] F. Shu, H. Lu, and Y. Ding, “Novel modbus adaptation method for IoT gateway,” *Proceedings of 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference, ITNEC 2019*. Chengdu, China, pp. 632–637, 2019, doi: 10.1109/ITNEC.2019.8729209.

[42] P. Shukla *et al.*, "Design and development of a MODBUS automation system for industrial applications," *2017 6th International Conference on Computer Applications in Electrical Engineering - Recent Advances, CERA 2017*, vol. 2018-Janua. IEEE, Roorkee, India, pp. 515–520, Oct-2018, doi: 10.1109/CERA.2017.8343383.

[43] V. Khedekar, S. Mahajan, and A. Karangale, "A Review on XBEE Technology," *International Journal of Emerging Technologies in Engineering Research (IJETER)*, vol. 4, no. 4. pp. 99–101, 2016.

[44] M. Khanafer, M. Guennoun, and H. T. Mouftah, "A survey of beacon-enabled IEEE 802.15.4 MAC protocols in wireless sensor networks," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 2. pp. 856–876, 2014, doi: 10.1109/SURV.2013.112613.00094.

[45] V. Boonsawat, "XBee wireless sensor networks for temperature monitoring," *Institute of Technology, Thammasat University Second Conference*. Pathum-Thani, Thailand, p. 6, 2010, doi: http://www2.siit.tu.ac.th/somsak/pub/final_XBeeWSN_100328.pdf.

[46] B. Ganeshprabu and M. Geethanjali, "Dynamic Monitoring and Optimization of Fault Diagnosis of Photo Voltaic Solar Power System Using ANN and Memetic Algorithm," *Circuits Syst.*, vol. 07, no. 11, pp. 3531–3540, 2016, doi: 10.4236/cs.2016.711300.

[47] H. Kumbhar, "Wireless sensor network using xbee on arduino platform an experimental study," *Proceedings - 2nd International Conference on Computing, Communication, Control and Automation, ICCUBEA 2016*. Institute of Electrical and Electronics Engineers Inc., Pune, India, 21-Feb-2017, doi: 10.1109/ICCUBEA.2016.7860081.

[48] Digi International, "XBee®/XBee-PRO S2C Zigbee®," *RF Module User Guide*. p. 309, 2018.

[49] C. Yang and H. R. Shao, "WiFi-based indoor positioning," *IEEE Communications Magazine*, vol. 53, no. 3. Institute of Electrical and Electronics Engineers Inc., pp. 150–157, 01-Mar-2015, doi: 10.1109/MCOM.2015.7060497.

[50] L. Yu and F. Zhang, "US20170347384A1 - Wifi connection method and wifi connection system for mobile terminal," 2017.

[51] Z. Zhao, X. Wu, X. Zhang, J. Zhao, and X. Y. Li, "ZigBee vs WiFi: Understanding issues and measuring performances of their coexistence," *2014 IEEE 33rd International Performance Computing and Communications Conference, IPCCC 2014*. IEEE, Austin, USA, pp. 1–8, Dec-2015, doi: 10.1109/PCCC.2014.7017082.

[52] Z. Zhou, C. Wu, Z. Yang, and Y. Liu, "Sensorless sensing with WiFi," *Tsinghua Science and Technology*, vol. 20, no. 1. Tsinghua University Press, pp. 1–6, 01-Feb-2015, doi: 10.1109/TST.2015.7040509.

[53] L. Li, X. Hu, K. Chen, and K. He, "The applications of WiFi-based Wireless Sensor Network in Internet of Things and Smart Grid," *Proceedings of the 2011 6th IEEE Conference on Industrial Electronics and Applications, ICIEA 2011*. Beijing, China, pp. 789–793, 2011, doi: 10.1109/ICIEA.2011.5975693.

[54] A. Patro, S. Govindan, and S. Banerjee, "Observing home wireless experience through WiFi APs," *Proceedings of the 19th annual international conference on Mobile computing & networking - MobiCom '13*. ACM Press, New York, New York, USA, p. 339, 2013, doi: 10.1145/2500423.2500448.

[55] M. Ahmed and Y.-C. Kim, “Hybrid Communication Network Architectures for Monitoring Large-Scale Wind Turbine,” *J Electr Eng Technol*, vol. 8, no. 6. pp. 1626–1636, 2013.
[56] M. E. Aminanto, P. D. Yoo, H. C. Tanuwidjaja, and K. Kim, “Weighted Feature Selection Techniques for Detecting Impersonation Attack in Wi-Fi Networks.” 2017.
[57] G. Mourya, S. Saxena, and S. Garhwal, “Water quality sensing with sensors and mobile smart phone,” Thapar University, 2015.
[58] T. Havinis, “US6219557B1 - System and method for providing location services in parallel to existing services in general packet radio services architecture,” 2001.
[59] R. J. Bates, *GPRS: General Packet Radio Service (Professional Telecom)*, 1st Editio. New York, New York, USA: McGraw-Hill, 2002.
[60] Z. Nawawi *et al.*, “Data transmission system of rotating electric field mill network using microcontroller and GSM module,” *Jurnal Teknologi (Sciences and Engineering)*, vol. 64, no. 4. pp. 109–112, 21-Oct-2013, doi: 10.11113/jt.v64.2110.
[61] N. Menon, A. Carlton, H. Wilk, and S. Howser, “US7039025B1 - System and method for providing general packet radio services in a private wireless network,” 2006.
[62] Q. Gu, C. Lu, F. Li, and C. Wan, “Monitoring dispatch information system of trucks and shovels in an open pit based on GIS/GPS/GPRS,” *Journal of China University of Mining and Technology*, vol. 18, no. 2. pp. 288–292, 2008, doi: 10.1016/S1006-1266(08)60061-9.
[63] S. Lee, G. Tewolde, and J. Kwon, “Design and implementation of vehicle tracking system using GPS/GSM/GPRS technology and smartphone application,” *2014 IEEE World Forum on Internet of Things, WF-IoT 2014*. pp. 353–358, 2014, doi: 10.1109/WF-IoT.2014.6803187.
[64] B. Wang, “Design of automatic meter reading system based on SCDMA,” *Dianli Xitong Baohu yu Kongzhi/Power System Protection and Control*, vol. 38, no. 4. pp. 103–105, 2010.
[65] V. Vujovic, M. Maksimovic, B. Perisic, and G. Milosevic, “A proposition of low cost Sensor Web implementation based on GSM/GPRS services,” *2015 IEEE 1st International Workshop on Consumer Electronics - Novi Sad, CE WS 2015*. Institute of Electrical and Electronics Engineers Inc., pp. 52–55, 28-Feb-2017, doi: 10.1109/CEWS.2015.7867154.
[66] G. Wang, J. Zhang, W. Li, D. Cui, and Y. Jing, “A forest fire monitoring system based on GPRS and ZigBee wireless sensor network,” *Proceedings of the 2010 5th IEEE Conference on Industrial Electronics and Applications, ICIEA 2010*. Taichung, Taiwan, pp. 1859–1862, 2010, doi: 10.1109/ICIEA.2010.5515417.
[67] H. Zhou, “GPRS based power quality monitoring system,” *Proceedings of 2005 IEEE Networking, Sensing and Control, ICNSC2005*, vol. 2005. pp. 496–501, 2005, doi: 10.1109/icnsc.2005.1461240.
[68] A. APOSTU, F. PUICAN, and G. ULARU, “Study on advantages and disadvantages of Cloud Computing–the advantages of Telemetry Applications in the Cloud.” 2013.
[69] S. Zahurul *et al.*, “Future strategic plan analysis for integrating distributed renewable generation to smart grid through wireless sensor network: Malaysia prospect,” *Renewable and Sustainable Energy Reviews*, vol. 53. pp. 978–992, 01-

Jan-2016, doi: 10.1016/j.rser.2015.09.020.
[70] M. Asadullah and A. Raza, “An overview of home automation systems,” *2016 2nd International Conference on Robotics and Artificial Intelligence, ICRAI 2016*. Institute of Electrical and Electronics Engineers Inc., Rawalpindi, Pakistan, pp. 27–31, 19-Dec-2016, doi: 10.1109/ICRAI.2016.7791223.
[71] A. Albagul, H. Efheij, and M. Muhammad, “Telemetry Home Automation Based on ZigBee,” *Libyan International Conference on Electrical Engineering and Technologies*, vol. LICEET. pp. 383–388, 2018.
[72] S. Maheshwari, K. Vasu, S. Mahapatra, and C. S. Kumar, “Measurement and Analysis of UDP Traffic over Wi-Fi and GPRS,” *Networking and Internet Architecture*, vol. 19, no. 11. Kharagpur, India, pp. 426–438, Jul-2017.
[73] C. Wang, K. Ren, W. Lou, and J. Li, “Toward publicly auditable secure cloud data storage services,” *IEEE Network*, vol. 24, no. 4. pp. 19–24, 2010, doi: 10.1109/MNET.2010.5510914.
[74] P. P. Ray, “A survey of IoT cloud platforms,” *Future Computing and Informatics Journal*, vol. 1, no. 1–2. Elsevier, pp. 35–46, 01-Dec-2016, doi: 10.1016/j.fcij.2017.02.001.
[75] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, “A gap analysis of Internet-of-Things platforms,” *Computer Communications*, vol. 89–90. pp. 5–16, 01-Sep-2016, doi: 10.1016/j.comcom.2016.03.015.
[76] B. de Bont, A. Souza, D. Johnathan, G. Aquino, R. Queiroz, and M. Melo, “Evaluating ThingSpeak as an IoT Event Platform on building a Smart Parking Application,” *Proceedings of the 11th Brazilian Symposium on Ubiquitous and Pervasive Computing*. SBC, Belém do Pará, Brazil, 12-Jul-2019.
[77] S. Salvi *et al.*, “Cloud based data analysis and monitoring of smart multi-level irrigation system using IoT,” *Proceedings of the International Conference on IoT in Social, Mobile, Analytics and Cloud, I-SMAC 2017*. IEEE, Palladam, India, pp. 752–757, Feb-2017, doi: 10.1109/I-SMAC.2017.8058279.
[78] P. Shekhar, M. Abebaw, M. Abebe Haile, M. Mehamed, and M. Kifle, “Temperature and Heart Attack Detection using IOT(Arduino and ThingSpeak),” *International Journal of Advances in Computer Science and Technology*, vol. 7, no. 11. pp. 75–82, 2018, doi: 10.30534/ijacst/2018/017112018.
[79] D. Nettikadan and S. Raj, “Smart Community Monitoring System using Thingspeak IoT Plaform,” *Article in International Journal of Applied Engineering Research*, vol. 13, no. October. Kerala, India, pp. 13402–13408, 2018.
[80] T. H. Nasution, M. A. Muchtar, S. Seniman, and I. Siregar, “Monitoring temperature and humidity of server room using Lattepanda and ThingSpeak,” *Journal of Physics: Conference Series*, vol. 1235, no. 1. IOP Publishing, p. 012068, Jun-2019, doi: 10.1088/1742-6596/1235/1/012068.
[81] P. Pounraj *et al.*, “Experimental investigation on Peltier based hybrid PV/T active solar still for enhancing the overall performance,” *Energy Conversion and Management*, vol. 168. pp. 371–381, 2018, doi: 10.1016/j.enconman.2018.05.011.
[82] R. M. Aliaga, J. A. Munoz, and M. A. Rojas, “Enhancing photovoltaic applications with embedded circuitry within the solar panels,” *2015 IEEE 13th Brazilian Power Electronics Conference and 1st Southern Power Electronics Conference, COBEP/SPEC 2016*. Institute of Electrical and Electronics Engineers Inc., Fortaleza, Brazil, 2015, doi: 10.1109/COBEP.2015.7420224.

[83] V. Pamadi and B. G. Nickerson, “Getting Started With 1-Wire Bus Devices,” Faculty of Computer Science University of New Brunswick, Fredericton, 2015.
[84] T. Topór-Kamiński, B. Krupanek, and J. Homa, “Delays models of measurement and control data transmission network,” *Studies in Computational Intelligence*, vol. 440. pp. 257–279, 2013, doi: 10.1007/978-3-642-31665-4_21.
[85] W. B. Van Der Merwe, P. E. Hertzog, and A. J. Swart, “Propagation Delays and Data Integrity of Cellular and WiFi Networks from IOT devices to cloud storage,” *Proceedings - 2019 Southern African Universities Power Engineering Conference/Robotics and Mechatronics/Pattern Recognition Association of South Africa, SAUPEC/RobMech/PRASA 2019*. Bloemfontein, South Africa, pp. 29–33, 2019, doi: 10.1109/RoboMech.2019.8704845.
[86] W. B. Van Der Merwe, P. E. Hertzog, and A. J. Swart, “Reliability and Transmission Delays of WiFi, GPRS and Radio Networks from IoT devices to cloud storage,” *Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2019*. Ballito, South Africa, pp. 20–25, Sep-2019.
[87] T. McBeath, “US7961637B2 - Method and apparatus for monitoring latency, jitter, packet throughput and packet loss ratio between two points on a network,” 2005.
[88] R. C. Perdana and F. W. Wibowo, “Quality of service for XBEE in implementation of wireless sensor network,” *Journal of Engineering and Applied Sciences*, vol. 11, no. 8. pp. 692–697, 2016.
[89] R. Muhendra, A. Rinaldi, M. Budiman, and Khairurrijal, “Development of WiFi Mesh Infrastructure for Internet of Things Applications,” *Procedia Engineering*, vol. 170. pp. 332–337, 2017, doi: 10.1016/j.proeng.2017.03.045.
[90] M. Cao and J. Fang, “Design of Remote Terminal of Air Compressor Based on STM32 and GPRS,” *IOP Conference Series: Materials Science and Engineering*, vol. 394. p. 032113, 2018, doi: 10.1088/1757-899X/394/3/032113.
[91] W. Goralski, *The illustrated network: How TCP/IP works in a modern network*. 2017.
[92] P. Kundur *et al.*, “Definition and Classification of Power System Stability IEEE/CIGRE Joint Task Force on Stability Terms and Definitions,” *IEEE Transactions on Power Systems*, vol. 19, no. 3. pp. 1387–1401, 2004, doi: 10.1109/tpwrs.2004.825981.

Annexure A: Sampler Unit Program

```
1   //*********************************************************************//
2   //https://electronics.stackexchange.com/questions/152380/wire-up-multiple-slave-arduino
    -devices-to-a-master-arduino-device    //
3   //
4   //
5   //
6   //
7   //*********************************************************************//
8
9   #include <TinyGPS++.h>
10  #include <LiquidCrystal.h>
11  #include <SPI.h>
12  #include <SD.h>
13  #include <OneWire.h>
14  #include <DallasTemperature.h>
15  #include <ModbusRtu.h>
16  #include <Scheduler.h>
17  #include <TimeAlarms.h>
18
19  #define serialBaud0 115200      // Computer port - 115200 port 0 - UNO / Mega
20  #define serialBaud1 9600        // GPS port - 9600 port 1 - Mega only
21  #define serialBaud2 19200       // Modbus Master port - 19200 port 2 - Mega only
22  #define serialBaud3 9600        // Not used yet - 115200 port 3 - Mega Only
23  #define portNum 2               // serial port number for Modbus Master
24  #define portSetuFp SERIAL_8N2// serial port setup for Modbus Master
25  #define pollTimeout 150         // 1000 Wait for reply from RTU
26  #define scanRate 10             // the scan rate between polls (100 min for SoftwareSerial)
27  #define masterAddress 0         // node id = 0 for master, = 1..247 for slave
28  #define hardwareType 0          // 0 for RS-232 and USB-FTDI or any pin number > 1 for RS-
    485
29  #define numberOfRegisters 12  // number of data elements
30
                                    #define iSlave_SlartAddress 0  // start Address of slave1 0 is n
    ot used for slave, allowcated to master setup
31  #define iSlave_Count 3  // number of slaves slaveArray. poll according to sequence of slaveArray
32
    #define iSampleCyclesToReset 0  // Number of Sample Cycles to Soft Reset. 0 = never Soft Reset
     (360)
33  #define pollsAfterSortResetInitiated 5  // Number of polls to poll RTU's after Reset was
    initiated to actual Soft Reset.
34
35  Scheduler scheduler = Scheduler();    //create a scheduler
36  //Scheduler scheduler2 = Scheduler();   //create a scheduler
37
38  // Scheduled Timers (t_)
39  int t_sequentialDataSample = 300;                       // (300) scheduler schedule for
    updating Data Sampling in Seconds
40  int t_selectDisplay = 750;      //scheduler schedule for updating Display Values
41  int t_setTemperature = 5000;  // scheduler schedule for updating Temperature
42  int t_heartBeat = 500;  // scheduler schedule for updating Heartbeat
43
44  long int softResetCounter = 0;
45  int pollsAfterSortResetInitiatedCounter = 0;
46
```

```
47  // data array for modbus network sharing
48  uint16_t au16data[numberOfRegisters];
49  uint8_t u8state;
50
51  // Modbuss Active Tx
52  int inPin = 3; // Modbus poll indicator
53
54  // Modbuss Slave Array
55  int slaveArry[iSlave_Count] = {1,2,3}; //, 3}; //{1, 2, 3} // addresses of slaves to be polled
56
57  /**
58  *      Modbus object declaration
59  *      u8id : node id = 0 for master, = 1..247 for slave
60  *      u8serno : serial port (use 0 for Serial)
61  *      u8txenpin : 0 for RS-232 and USB-FTDI
62  *      or any pin number > 1 for RS-485
63  */
64  Modbus master(masterAddress,portNum,hardwareType); // this is master and RS-232 or
    USB-FTDI via software serial
65
66  /**
67  * This is an structe which contains a query to an slave device
68  */
69  modbus_t telegram;
70  unsigned long u32wait;
71
72  // Data wire is plugged into pin 2 on the Arduino
73  #define ONE_WIRE_BUS 2
74  /*********************************************************************/
75  // Setup a oneWire instance to communicate with any OneWire devices
76  // (not just Maxim/Dallas temperature ICs)
77  OneWire oneWire(ONE_WIRE_BUS);
78  /*********************************************************************/
79  // Pass our oneWire reference to Dallas Temperature.
80  DallasTemperature sensors(&oneWire);
81  /*********************************************************************/
82
83  /********** SD Card Pin Allocation and File Assignment *************/
84  const int chipSelect = 53;      //10 Werk op UNO en 53 werk op die Mega
85  long int myIndex = 0;
86  File dataFile;
87  File indexFile;
88  File mainLogFile;
89  String myReadString;
90
91  /***************** Temperature String Assignmentn ****************/
92  String myTemperature;
93
94  /************************ LCD Pin Allocation *********************/
95  LiquidCrystal lcd(27, 26, 25, 24, 23, 22);                    // initialize the library with the
    numbers of the interface pins
96
97  /********************* GPS String Assignment *********************/
98  String gpsLogYear = "";
99  String gpsLogMonth = "";
100 String gpsLogDay = "";
101 String gpsLogHour = "";
102 String gpsLogMinute = "";
```

```
String gpsLogSecond = "";
String gpsLogTime = "0:0:0";
String gpsLogDate = "0/0/0";
String gpsSats = "0";

String LcdYear = "";
String LcdMonth = "";
String LcdDay = "";
String LcdHour = "";
String LcdMinute = "";
String LcdSecond = "";
String LcdTime = "0:0:0";
String LcdDate = "0/0/0";

String gpsYear = "";
String gpsMonth = "";
String gpsDay = "";
String gpsHour = "";
String gpsMinute = "";
String gpsSecond = "";
String gpsTime = "0:0:0";
String gpsDate = "0/0/0";
String gpsLat = "";
String gpsLng = "";
String gpsAlt = "";
String gpsSpeed = "";
String gpsDirection = "";

//long int samplePeriod= 0;

int voltageSensorPin = 6;      // select the input pin for the voltage sensing
int ampSensorPin = 7;          // select the input pin for the amps sensing

//float f_voltageSensorValue = 0;
//float f_ampSensorValue = 0;

long int voltageSensorValue = 0;
long int ampSensorValue = 0;
float voltPerCount = 0.488;
float voltageFactor = 5; // 5V = 25V
//float shuntResistorValue = 2; // Rs = 2ohm
int i = 0;

//String voltageStringValue = "";
//String ampStringValue = "";


/****************** Smart Delay Definitions ********************/
//unsigned long smartdelay_1_previousMillis = 0;
//unsigned long smartdelay_2_previousMillis = 0;
//unsigned long smartdelay_3_previousMillis = 0;
//unsigned long smartdelay_4_previousMillis = 0;

int inPinDisplay0 = 8;          // Display selector switch bit value 1
int inPinDisplay1 = 9;          // Display selector switch bit value 2

int clearLCD1 = 0;
int clearLCD2 = 0;
```

```
161  int clearLCD3 = 0;
162  int clearLCD4 = 0;
163
164  //int dateCheck = 0;
165
166  // The TinyGPS++ object
167  TinyGPSPlus gps;
168
169  /********************* Smart Blink Declaration ********************/
170  //int ledState = LOW;
171  //unsigned long Blink_previousMillis = 0;
172  //const long Blink_interval = 500;
173
174  const int blinkPin1 = 45;
175  boolean blink1State = false;
176  /******************* End Smart Blink Declaration ******************/
177
178
179
180
181
182  void setup() {
183  Serial.begin(serialBaud0);          // Computer comms
184  while (!Serial) {
185  ; // wait for serial port to connect. Needed for Leonardo only
186  }
187  Serial1.begin(serialBaud1); // GPS comms
188
189  // Modbus Parameters
190  master.begin( serialBaud2, portSetup ); // begin the ModBus object.
     The first parameter is the address of yourSoftwareSerial address.
      Do not forget the "&". 9600 means baud-rate at 9600
191  master.setTimeOut( pollTimeout ); // if there is no answer in 2000 ms, roll over
192  u32wait = millis() + 1000;
193  u8state = 0;
194
195  pinMode(inPin, OUTPUT);       // Modbus poll indicator pin-4
196  pinMode(inPinDisplay0, INPUT); // Select Display 0 pin-8
197  pinMode(inPinDisplay1, INPUT); // Select Display 1 pin-9
198
199  lcd.begin(16, 2);                  // set up the LCD's number of columns and rows:
200  lcd.print("Soft Reset");           // Print a message to the LCD.
201  delay(200);
202  lcd.clear();
203
204
205  /********************** SD Card Setup Start **********************/
206  Serial.print("Initializing SD card...");
207  // make sure that the default chip select pin is set to
208  // output, even if you don't use it:
209  pinMode(SS, OUTPUT);
210
211  // see if the card is present and can be initialized:
212  if (!SD.begin(chipSelect)) {
213  Serial.println("Card failed, or not present");
214  // don't do anything more:
215  return; //while (1) ;
216  }
```

```
217   Serial.println("card initialized.");
218
219   // GPS SD card method from Jerry Blum
220   indexFile = SD.open("Index.txt");
221   if (indexFile) {
222   Serial.println("Reading Index.txt ..... ");
223
224   // read from the file until there's nothing else in it:
225   while (indexFile.available()) {
226   char x(indexFile.read());
227   myReadString += x;
228   }
229
230   Serial.println(myReadString);
231   Serial.println(myReadString.toInt());
232   myIndex = myReadString.toInt();
233   myIndex += 1;
234   // close the file:
235   indexFile.close();
236   } else {
237   // if the file didn't open, print an error:
238   Serial.println("error opening test.txt");
239   }
240   indexFile = SD.open("Index.txt", FILE_WRITE);
241
242
243   /********* Create New LOG file with a restart method ***********/
244   //            char filename[] = "DATA0000.CSV";
245   //            for (uint8_t i = 0; i < 10000; i++) {
246   //              filename[6] = i/10 + '0';
247   //              filename[7] = i%10 + '0';
248   //              if (! SD.exists(filename)) {
249   //                // only open a new file if it doesn't exist
250   //                dataFile = SD.open(filename, FILE_WRITE);
251   //                break;       // leave the loop!
252   //                }
253   //            }
254   //
255   //            Serial.print("Logging to: ");
256   //            Serial.println(filename);
257
258   /********* Append LOG file with a restart method **************/
259   // Open up the file we're going to log to!
260   mainLogFile = SD.open("MainLog.CSV", FILE_WRITE);
261   if (! mainLogFile) {
262   Serial.println("error opening MainLog.CSV");
263   // Wait forever since we cant write data
264   return; //while (1) ;
265   }
266
267   Serial.println("Logging to: MainLog.CSV");
268   //Serial.println(mainLogFile);
269
270   mainLogFile = SD.open("MainLog.CSV", FILE_WRITE);
271
272
273   /*
274   // Open up the file we're going to log to!
```

```
275  indexFile = SD.open("Index.txt", FILE_WRITE);
276  if (! indexFile) {
277  Serial.println("error opening index.txt");
278  // Wait forever since we cant write data
279  while (1) ;
280  }
281  */
282
283  /************************ End SD Card Setup ************************/
284
285  /************************ Smart Blink Setup ************************/
286  pinMode(LED_BUILTIN, OUTPUT);                    // LED_BUILTIN is set to the correct LED
     pin independent of which board is used.
287  digitalWrite(LED_BUILTIN, LOW);
288  /************************ End Smart Blink Setup ********************/
289
290
291  /************************ Scheduler Routines ************************/
292  //scheduler.schedule(sequentialDataSample, t_sequentialDataSample);
293  scheduler.schedule(selectDisplay, t_selectDisplay);
294  scheduler.schedule(getTemperature, t_setTemperature);
295  scheduler.schedule(heartBeat, t_heartBeat);
296  /************************ End Scheduler Routin ********************/
297
298  /******************* Repeat Routines - Long Periods ********************/
299  //setTime(8,29,0,1,1,10); // set time to 8:29:00am Jan 1 2010
300  Alarm.timerRepeat(t_sequentialDataSample, sequentialDataSample); //timer for every5minutes
301  /****************** End Repeat Routines - Long Periods ***************/
302
303  }
304
305
306  /*****************************************************************************/
307  /************************ Soft Reset *********************************/
308  void(* resetFunc) (void) = 0; //declare reset function at address 0
309  // resetFunc(); //Call Reset
310  /************************ End Soft Reset *****************************/
311
312
313  void loop() {
314  //Serial.println("Main loop");
315
316  CheckForGpsSerialData();
317  scheduler.update();
318  pollMySlaves();
319  au16data[11] = 0; // If = 1 Sort Reset RTU Slaves
320  au16data[0] = 0; // No New Modbus data available
321  softReset();
322  GetAnalogueValues();
323  Alarm.delay(1);
324  }
325
326
327
328
329
330  /*****************************************************************************/
331  /*****************************************************************************/
```

```
/*****************************************************************************/
/*****************************************************************************/


/*****************************************************************************/
void getTemperature()
{
sensors.requestTemperatures(); // Send the command to get temperature readings
myTemperature = (sensors.getTempCByIndex(0));
scheduler.schedule(getTemperature, t_setTemperature);
//Serial.println("Get Temperature");
}

/*****************************************************************************/
static void sequentialDataSample()
{
gpsLogDateTime();
GetAnalogueValues();
SD_CardWriteData();
au16data[0] = 1; // New Modbus data available
//samplePeriod = 5000;            //Set Sample Period here.                    300000
Serial.println(gpsLogTime) + (" ") + (gpsLogDate) + (" ") + (LcdTime) + (" ") + (myTemperature);
Serial.println("SequentialDataSample ***************************************");
//blink1State = !blink1State;
//digitalWrite(blinkPin1, blink1State);

//scheduler.schedule(sequentialDataSample, t_sequentialDataSample);
softResetCounter ++;
i++;
}

/*****************************************************************************/
void SD_CardWriteData()
{
String dataString = "";
dataString = myIndex;
dataString += ",";

dataString += i;
dataString += ",";

dataString += gpsLogDate;
dataString += ",";

dataString += gpsLogTime;
dataString += ",";

dataString += "Sats";
dataString += ",";

dataString += gpsSats;
dataString += ",";

dataString += "Temp = ";
dataString += ",";

dataString += myTemperature;
```

```
390   dataString += ",";
391
392   dataString += "Voltage = ";
393   dataString += ",";
394
395   dataString += voltageSensorValue;  //voltageStringValue;
396   dataString += ",";
397
398   dataString += "Amp = ";
399   dataString += ",";
400
401   dataString += ampSensorValue;  //ampStringValue;
402
403   Serial.println(dataString); //*********************************************
404   //Serial.println(myIndex);
405
406   /********************  SD  Write  -  Method  1  ***********************/
407   //  dataFile.println(dataString);
408   mainLogFile.println(dataString);
409
410   indexFile.seek(0);  // OverWrite indexFile all the time
411   indexFile.print(myIndex);
412
413   //  dataFile.flush();
414   mainLogFile.flush();
415   indexFile.flush();
416
417   myIndex++;
418
419   }
420
421   /*************************************************************/
422   void  gpsLogDateTime(){
423   gpsLogYear  =  String(gps.date.year());
424   gpsLogMonth  =  String(gps.date.month());
425   gpsLogDay  =  String(gps.date.day());
426   gpsLogDate  =  gpsLogYear  +  "/"  +  gpsLogMonth  +  "/"  +  gpsLogDay;
427
428   gpsLogHour  =  String(gps.time.hour());
429   gpsLogMinute  =  String(gps.time.minute());
430   gpsLogSecond  =  String(gps.time.second());
431   gpsLogTime  =  gpsLogHour  +  ":"  +  gpsLogMinute  +  ":"  +  gpsLogSecond;
432
433   gpsSats  =  String(gps.satellites.value());
434   }
435
436   /*************************************************************/
437   /*****************  Select  Display  **************************/
438   void  selectDisplay()
439   {
440
441
442   if  ((digitalRead(inPinDisplay0)  ==  0)  &&  digitalRead(inPinDisplay1)  ==  0)
443   {
444   if  (clearLCD1  ==  0)
445   {
446   clearLCD1  =  1;
447   clearLCD2  =  0;
```

```
448   clearLCD3 = 0;
449   clearLCD4 = 0;
450   lcd.clear();
451   }
452   //smartdelay_1(1000);
453   Display1();
454   }
455
456
457   if ((digitalRead(inPinDisplay0) == 0) && digitalRead(inPinDisplay1) == 1)
458   {
459   if (clearLCD2 == 0)
460   {
461   clearLCD1 = 0;
462   clearLCD2 = 1;
463   clearLCD3 = 0;
464   clearLCD4 = 0;
465   lcd.clear();
466   }
467   //smartdelay_2(1000);
468   Display2();
469   }
470
471
472   if ((digitalRead(inPinDisplay0) == 1) && digitalRead(inPinDisplay1) == 0)
473   {
474   if (clearLCD3 == 0)
475   {
476   clearLCD1 = 0;
477   clearLCD2 = 0;
478   clearLCD3 = 1;
479   clearLCD4 = 0;
480   lcd.clear();
481   }
482   //smartdelay_3(1000);
483   lcd.setCursor(0, 0);
484   lcd.print("Display 3");
485   }
486
487
488   if ((digitalRead(inPinDisplay0) == 1) && digitalRead(inPinDisplay1) == 1)
489   {
490   if (clearLCD4 == 0)
491   {
492   clearLCD1 = 0;
493   clearLCD2 = 0;
494   clearLCD3 = 0;
495   clearLCD4 = 1;
496   lcd.clear();
497   }
498   //smartdelay_4(1000);
499   lcd.setCursor(0, 0);
500   lcd.print("Display 4");
501   delay(2500);
502   resetFunc();
503   }
504   scheduler.schedule(selectDisplay, t_selectDisplay);
505   }
```

```
506
507
508  /********************  Smart  Delay  1  *************************/
509  static  void  Display1()
510  {
511  LcdDateTime();
512
513  lcd.setCursor(0,  0);
514  lcd.print("                              ");
515  lcd.setCursor(0,  1);
516  lcd.print("                              ");
517
518  lcd.setCursor(0,  0);
519  lcd.print(LcdTime);
520  lcd.setCursor(10,0);
521  lcd.print(LcdDate);
522
523  lcd.setCursor(0,  1);
524  lcd.print(gpsLogTime);
525  lcd.setCursor(10,1);
526  lcd.print("Temp:");
527  lcd.print(myTemperature);
528
529  //  Serial.println((LcdTime) + (" ") + (gpsLogDate) + (" - Sample Period = ") +
     (samplePeriod) + (" - ") +(gps.time.second()));  // ************************
530
531  //Serial.println("Display1  *****************");
532  //scheduler.schedule(Display1,  1000);
533
534  //i++;
535
536  //SmartFlipFlop();
537  //  }
538  }
539
540  /********************  Smart  Delay  2  ************************/
541  static  void  Display2()
542  {
543  //  unsigned  long  smartdelay_2_currentMillis  =  millis();
544  //  if  (smartdelay_2_currentMillis  -  smartdelay_2_previousMillis  >=  ms2)
545  //              {
546  //                smartdelay_2_previousMillis  =  smartdelay_2_currentMillis;
547  lcd.setCursor(0,  0);
548  lcd.print("              ");
549  lcd.setCursor(0,  1);
550  lcd.print("              ");
551
552  lcd.setCursor(0,  0);
553  lcd.print(gpsAlt);
554  lcd.setCursor(10,0);
555  lcd.print(gpsSats);
556  lcd.setCursor(0,  1);
557  lcd.print(gpsSpeed);
558  lcd.setCursor(10,1);
559  lcd.print(gpsDirection);
560
561  //scheduler.schedule(Display2,  1000);
562
```

```
563   //              i++;
564
565   //            }
566   }
567
568   /*****************************************************************/
569   void  LcdDateTime(){
570   LcdYear  =  String(gps.date.year());
571   LcdMonth  =  String(gps.date.month());
572   LcdDay  =  String(gps.date.day());
573   LcdDate  =  LcdYear  +  "/"  +  LcdMonth  +  "/"  +  LcdDay;
574
575   LcdHour  =  String(gps.time.hour());
576   LcdMinute  =  String(gps.time.minute());
577   LcdSecond  =  String(gps.time.second());
578   LcdTime  =  LcdHour  +  ":"  +  LcdMinute  +  ":"  +  LcdSecond;
579
580   gpsSats  =  String(gps.satellites.value());
581   }
582
583   /*****************************************************************/
584   void  GetAnalogueValues(){
585   long  int  ampSensorValueTemp  =  0;
586   int  SamplesToAverage  =  30;
587   ampSensorValue  =  0;
588
589   for  (int  a=0;  a  <=  SamplesToAverage;  a++)
590   {
591   ampSensorValueTemp  =(((analogRead(ampSensorPin) -  511) *  4882812) /  1023) /  185;
592   ampSensorValue  +=  ampSensorValueTemp;
593   }
594
595   ampSensorValue  =              ampSensorValue  /  SamplesToAverage;
596   voltageSensorValue  =  analogRead(voltageSensorPin) *  voltPerCount  *  voltageFactor;
597
598   Serial.print("Volts = ");
599   Serial.print(voltageSensorValue);
600   Serial.print("   ");
601   Serial.print("Raw Amps = ");
602   Serial.print(analogRead(ampSensorPin));
603   Serial.print("   ");
604   Serial.print("Amps = ");
605   Serial.print(ampSensorValue);
606
607   //  voltageStringValue  =  voltageSensorValue;
608   //  ampStringValue  =  ampSensorValue;
609
610   //  Serial.print("                    ");
611   //  Serial.print(voltageStringValue);
612   //  Serial.print("                    ");
613   //  Serial.println(ampStringValue);
614
615   /*********************  Old  Code  ************************************
616   voltageSensorValue  =  analogRead(voltageSensorPin)  *  voltPerCount  *  voltageFactor;
617   ampSensorValue  =  (analogRead(ampSensorPin)  *  voltPerCount)  /  shuntResistorValue;
618
619   Serial.print("Volts  =  ");
620   Serial.print(voltageSensorValue);
```

```
621   Serial.print("                    ");
622   Serial.print("Amps = ");
623   Serial.println(ampSensorValue);
624
625   //voltageSensorValue = random(1,1024); // generate random number between 1 & 5
      (minimum is inclusive, maximum is exclusive)
626   //ampSensorValue = random(1,1024); // generate random number between 1 & 5
      (minimum is inclusive, maximum is exclusive)
627
628
629   voltageStringValue = voltageSensorValue;
630   ampStringValue = ampSensorValue;
631
632   //Serial.println("Voltage Value = " + voltageStringValue + (" ") + voltageSensorValue);
633   //Serial.println("Amp Value = " + ampStringValue + (" ") + ampSensorValue);
634   ***************************** End of Old Code *****************************/
635   }
636   /*****************************************************************************/
637
638   /*****************************************************************************/
639
640   /*****************************************************************************/
641   //***************************************************************************
642   void updateModbusTxRegisterData(){
643
644   float modbusTemp;
645   modbusTemp = myTemperature.toFloat() * 100;
646
647   //au16data[0] = 0;//counter;
648   au16data[1] = myIndex - 1; //1;//counter;
649   au16data[2] = gps.date.year(); //LcdYear.toInt(); //2;//counter;
650   au16data[3] = gps.date.month(); //LcdMonth.toInt(); //3;//counter;
651   au16data[4] = gps.date.day(); //LcdDay.toInt(); //4;//counter;
652   au16data[5] = gps.time.hour(); //LcdHour.toInt(); //5;//counter;
653   au16data[6] = gps.time.minute(); //LcdMinute.toInt(); //6;//counter;
654   au16data[7] = gps.time.second(); //LcdSecond.toInt(); //7;//counter;
655   au16data[8] = voltageSensorValue; //8;//counter;
656   au16data[9] = ampSensorValue; //9;//counter;
657   au16data[10] = modbusTemp; //10;//counter;
658
659   /*gpsLogYear = String(gps.date.year());
660   gpsLogMonth = String(gps.date.month());
661   gpsLogDay = String(gps.date.day());
662   gpsLogDate = gpsLogYear + "/" + gpsLogMonth + "/" + gpsLogDay;
663
664   gpsLogHour = String(gps.time.hour());
665   gpsLogMinute = String(gps.time.minute());
666   gpsLogSecond = String(gps.time.second());*/
667
668
669   }
670   /*****************************************************************************/
671   void pollMySlaves()
672   {
673   int idx = iSlave_SlartAddress;
674   uint8_t localstate = 0;
675   updateModbusTxRegisterData();
676
```

```
677   telegram.u8fct = 16; // function code (this one is registers read)
678   telegram.u16RegAdd = 0; // start address in slave
679   telegram.u16CoilsNo = numberOfRegisters; // number of elements (coils or registers) to read
680   telegram.au16reg = au16data; // pointer to a memory array in the Arduino -
      updateModbusTxRegisterData()
681
682   while (idx < iSlave_SlartAddress + iSlave_Count)
683   {
684   telegram.u8id = slaveArry[idx];
685   localstate = 0;
686   while(true)
687   {
688   switch( localstate ) {
689   case 0:
690   if (millis() > u32wait) {
691   localstate++;
692   //Serial.println("State 0: Waited");
693   }// wait state
694   break;
695   case 1:
696   master.query( telegram ); // send query (only once)
697   //Serial.println("State 1: Sent to RTU");
698   digitalWrite(inPin, HIGH);
699   localstate++;
700   break;
701   case 2:
702   master.poll(); // check incoming messages
703   if (master.getState() == COM_IDLE) {
704   u32wait = millis() + scanRate;
705   //Serial.println("State 2: COM IDLE");
706   digitalWrite(inPin, LOW);
707   goto exit_loop;
708   }
709   break;
710   }
711   }
712   exit_loop: ;
713
714   idx++;
715   }
716   //scheduler.schedule(pollMySlaves, 500);
717   }
718   /*****************************************************************/
719   void CheckForGpsSerialData()
720   {
721   while (Serial1.available() > 0)
722   {
723   //if (gps.encode(Serial1.read()))
724   gps.encode(Serial1.read());
725   }
726   // Serial.println("Check for GPS Data");
727   // scheduler.schedule(CheckForGpsSerialData, 100);
728   }
729
730   /*****************************************************************/
731   void heartBeat()
732   {
733   blink1State = !blink1State;
```

```
734   digitalWrite(blinkPin1,  blink1State);
735   scheduler.schedule(heartBeat,  t_heartBeat);
736
737   //selectDisplay();
738   }
739
740   /*****************************************************************/
741   void  softReset()
742   {
743   if  (iSampleCyclesToReset  >  0  )
744   {
745   if  (iSampleCyclesToReset  ==  softResetCounter)
746   {
747   pollsAfterSortResetInitiatedCounter  ++;
748   }
749   if  ((pollsAfterSortResetInitiated  -  1) ==  pollsAfterSortResetInitiatedCounter)
750   {
751   au16data[11] =  1;
752   Serial.println("Sending Reset to RTU's");
753   }
754   if  (pollsAfterSortResetInitiated  ==  pollsAfterSortResetInitiatedCounter)
755   {
756   Serial.println("Reset");
757   Serial.println("");
758   delay(100);
759   resetFunc();
760   }
761   }
762
763   }
764   /*****************************************************************/
```

Annexure B: WiFi Unit Program

```
1   //*************************************************************************//
2   //  https://www.instructables.com/id/Send-sensor-data-DHT11-BMP180-to-ThingSpeak-with-a/      //
3   //  https://www.instructables.com/id/Getting-Started-With-the-ESP8266-ESP-01/                 //
4   //  https://room-15.github.io/blog/2015/03/26/esp8266-at-command-reference/                  //
5   //  https://github.com/itead/ITEADLIB_Arduino_WeeESP8266                                     //
6   //  http://fab.cba.mit.edu/classes/863.14/tutorials/Programming/serialwifi.html              //
7   //                                                                                           //
8   //*************************************************************************//
9
10  #include <ModbusRtu.h>
11  //#include <SoftwareSerial.h>
12  //#include <LiquidCrystal.h>
13
14  /********************** Define Modbus Parameters *********************/
15  #define serialBaud 115200 // 115200 Monitor port
16  #define serialBaud1 115200          // 115200 WiFi Port
17  #define serialBaud2 19200 // 115200 Modbus Port
18  #define serialBaud3 19200 // 115200 Modbus Port
19  #define ModbusPortNum 3 // serial port number for Modbus Master
20  #define portSetup SERIAL 8N2 // serial port setup for Modbus Master
21  #define hardwareType 0 // 0 for RS-232 and USB-FTDI or any pin number > 1 for RS-485
22  //#define ssBaud 19200 // SoftwareSerial Bit rate 8N2
23  //#define ssRxPin 6 // SoftwareSerial Rx Pin
24  //#define ssTxPin 7 // SoftwareSerial Tx Pin
25  #define slaveAddress 2 // node id = 0 for master, = 1..247 for slave
26  //#define hardwareType 0 // 0 for RS-232 and USB-
    FTDI or any pin number > 1 for RS-485
27  #define numberOfRegisters 12
28  #define i_AT_ErrorRetry 3 // Number of Retries when Error is Returned
29  #define i_DeadModemRetry 3 // Number of Retries when Dead Modem is Detected
30
31
32
33  uint16_t au16data[numberOfRegisters] = {0,0,0,0,0,0,0,0,0,0,0,0}; // data array for modbus
34
35  Modbus slave(slaveAddress,ModbusPortNum,hardwareType);
36
37  //SoftwareSerial mySerial(ssRxPin, ssTxPin);//Rx-Tx - Create a SoftwareSerial object.
38
39  //char ssid[] = "MyMobileWiFi";   //        your network SSID (name)
40  char ssid[] = "MyWiFi";    //        your network SSID (name)
41  char pass[] = "0000000000";        // your network password
42
43  /********************** Declare Data Integers ************************/
44  int newData = 0;
45  int softReset = 0;
46  long int dataIndex = 0;
47  int dataYear = 0;
48  int dataMonth = 0;
49  int dataDay = 0;
50  int dataHour = 0;
51  int dataMinute = 0;
52  int dataSecond = 0;
53  int RawDataTemperature = 0;
```

```
54  int RawDataAmp = 0;
55  int RawDataVolt = 0;
56  float dataTemperature = 0;
57  float dataAmp = 0;
58  float dataVolt = 0;
59
60  int AT_ErrorRetryCounter = 1;
61  //int fieldValue_1 = 10;
62  char fieldValue_1[12] = "";
63  int fieldValue_2 = 20;
64  int fieldValue_3 = 30;
65  int fieldValue_4 = 40;
66  int fieldValue_5 = 50;
67  int fieldValue_6 = 60;
68  int fieldValue_7 = 70;
69  int fieldValue_8 = 0;
70  //int rampValue = 1;
71  int i_StateMachine_State = 1;
72  int i_startATmillis = 0;
73  int i_endATmillis = 0;
74  int i_StateMachineMillis = 0;
75
76  //int inPin = 8;
77
78  String dataDate = "";
79  String dataTime = "";
80  String sdDataString = "";
81
82  bool b_StateMachine = false;
83
84  char PingIP[] = "8.8.8.8";
85  char RouterIP[15] ="";
86  //char GSM_RouterIP[] = "192.168.8.1";
87  //char ADSL_RouterIP[] = "192.168.1.1";
88  char AT_String[50];
89
90  int i = 0;
91
92  char aux_str[200];
93  char ip_data[1024];
94  char ThingSpeakKey[] = "GK5J5W0HJ75P98T8";
95
96
97  /************************ LCD Pin Allocation ***************************/
98  //LiquidCrystal lcd(27, 26, 25, 24, 23, 22);  // initialize the library with the numbers of the interface pins
99
100 void setup() {
101 /************************ Serial Port Setup ***************************/
102 Serial.begin(serialBaud);  // USB/Com0 Monitor Port
103 while (!Serial) {
104 ;  // wait for serial port to connect. Needed for Leonardo only
105 }
106
107 /************************ Serial for Modbus Setup ************************/
108 slave.begin( serialBaud3, portSetup );  // begin the ModBus object.
109 //slave.begin( mySerial, portSetup );  // begin the ModBus object.
110
111 /******************************* LCD Setup ******************************/
```

```
112  //lcd.begin(16, 2); // set up the LCD's number of columns and rows:
113  //lcd.print("Data Logger Ver5.5");  // Print a message to the LCD.
114  //delay(2000);
115  //lcd.clear();
116  /********************************** ESP Setup ****************************/
117  pinMode(9, OUTPUT);
118  Serial1.begin(serialBaud1);
119
120  if (sendATcommand2("AT", "OK", "ERROR", 1000) == 1)
121  {
122  Serial.println("ESP8266 present and OK");
123  Serial.println("Now Setting Mode ..... ");
124  SetEspMode();
125  Serial.println("Now Connecting to WiFi");
126  WiFiConnect();
127  }
128  else
129  {
130  Serial.println("ESP8266 Error");
131  }
132
133  Serial.println();
134
135  Serial.println("Your IP Information: ");
136  sendATcommand2("AT+CIFSR", "OK", "ERROR", 1000);
137  delay(2000);
138
139
140
141
     // Select whitch router to ping for connectivity check (CheckWiFiRouterConnection) from SSID infor
     mation.
142  if (strstr(ssid, "MyMobileWiFi") != NULL)
143  {
144  sprintf(RouterIP, "192.168.8.1");
145  }
146  else if (strstr(ssid, "MyWiFi") != NULL)
147  {
148  sprintf(RouterIP,"192.168.1.1");
149  }
150  Serial.print("Router IP to ping: ");
151  Serial.println(RouterIP);
152
153
154  }
155
156  /************************ Soft Reset ******************************/
157  void(* resetFunc) (void) = 0;//declare reset function at address 0
158  // resetFunc(); //Call Reset
159  /********************** End Soft Reset ****************************/
160
161
162  /*****************************************************************/
163  void loop() {
164  updateSlaveValues();
165  newDataAvailable();
166
167  // dataIndex = 100;
```

```
// i = 200;
// sprintf(dataIndex, ".", i);
// Serial.println(dataIndex);

//softResetRTU();
//char x = strncat("abc", (strncat("." ,"d", 1)), 5);

}

/************************************************************************/




/************************************************************************/
void updateSlaveValues() {
slave.poll( au16data, 12 );

newData = au16data[0];
dataIndex = au16data[1];
dataYear = au16data[2];
dataMonth = au16data[3];
dataDay = au16data[4];
dataHour = au16data[5];
dataMinute = au16data[6];
dataSecond = au16data[7];
RawDataVolt = au16data[8];
RawDataAmp = au16data[9];
RawDataTemperature = au16data[10];
softReset = au16data[11];

//dataVolt = RawDataVolt / 100.0;
//dataAmp = RawDataAmp / 100.0;
//dataTemperature = RawDataTemperature / 100.0;
// Serial.print(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>            ");
// Serial.println(String(dataVolt));

structureDateTime();
buildDataString();



}
/************************************************************************/
void newDataAvailable() {

if(newData == 1)
{
au16data[0] = 0;
Serial.println(sdDataString);
Serial.println(("Volt = ") + String(au16data[8]) + ("   Amp = ") +String(au16data[9]) +
 ("            Temp = ")+String(au16data[10]));
StateMachine();
Serial.println("End of State Machine. Exit Loop. newData = " + String(newData));
i++;
}
}
```

```
225  /*********************************************************************/
226
227  /*********************************************************************/
228  void buildDataString() {
229  sdDataString = "";
230  sdDataString += newData;
231
232  sdDataString += ",";
233  sdDataString += dataIndex;
234
235  sdDataString += ",";
236  sdDataString += dataDate;
237
238  sdDataString += ",";
239  sdDataString += dataTime;
240
241  sdDataString += ",";
242  sdDataString += "Temp = ";
243  sdDataString += ",";
244  sdDataString += String(dataTemperature);
245
246  sdDataString += ",";
247  sdDataString += "Voltage = ";
248  sdDataString += ",";
249  sdDataString += String(dataVolt);
250
251  sdDataString += ",";
252  sdDataString += "Amp = ";
253  sdDataString += ",";
254  sdDataString += String(dataAmp);
255
256  //Serial.println(sdDataString);
257  }
258  /*********************************************************************/
259  void structureDateTime(){
260
261  dataDate ="";
262  dataTime ="";
263
264  dataDate = dataDay;
265  dataDate += "/";
266  dataDate += dataMonth;
267  dataDate += "/";
268  dataDate += dataYear;
269
270  dataTime = dataHour;
271  dataTime += ":";
272  dataTime += dataMinute;
273  dataTime += ":";
274  dataTime += dataSecond;
275
276  //Serial.print(dataDate);
277  //Serial.print("  ");
278  //Serial.println(dataTime);
279  }
280
281  /*********************************************************************/
282  /*void softResetRTU()
```

```
283   {
284   if(softReset == 1)
285   {
286   Serial.println("Soft Reset");
287   delay(500);
288   resetFunc();
289   }
290   }
291   /*****************************************************************************/
292   //*****************************************************************************//
293   //****************************** State 1 ***********************************//
294   void CheckConnection(){
295   i_startATmillis = millis();
296
297   if (CheckWiFiRouterConnection() == true)
298   {
299   Serial.println("Connected to Router");
300   i_StateMachine_State = 2;
301   Serial.println("Going from State 1 to State 2");
302   Serial.println();
303   }
304   else
305   {
306   Serial.println("No Reply from Router - Check Router");
307   SetEspMode();
308   WiFiConnect();
309   i_StateMachine_State = 1;
310   Serial.println("Remaining in State 1");
311   Serial.println();
312   }
313
314   }
315   //****************************** State 2 ***********************************//
316   void SendData(){
317   // Opens a TCP socket
318   if (sendATcommand2("AT+CIPSTART=\"TCP\",\"api.thingspeak.com\",80\r\n", "CONNECT",
        "CONNECT FAIL", 3000) == 1)
319   {
320   Serial.println("Connected");
321   Serial.println("TCP socket open to ThingSpeak");
322   GetThingSpeakFieldValues();
323   sprintf(ip_data, "GET/update?api_key=%s&field1=%s&field2=%d&field3=%d&field4=%d
      &field5=%d&field6=%d&field7=%d&field8=%dHTTP/1.1\r\nHost: api.thingspeak.com
      \r\nConnection: keep-alive\r\n\r\n",
324   ThingSpeakKey,
325   fieldValue_1,
326   fieldValue_2,
327   fieldValue_3,
328   fieldValue_4,
329   fieldValue_5,
330   fieldValue_6,
331   fieldValue_7,
332   fieldValue_8);
333   //fieldValue_8 = i_StateMachineMillis);
334   //Serial.println(ip_data);
335
336   sprintf(aux_str,"AT+CIPSEND=%d", strlen(ip_data));
337   Serial.println(aux_str);
```

```
338   if (sendATcommand2(aux_str, ">", "ERROR", 10000) == 1)
339   {
340   // delay(1500);
341   sendATcommand2(ip_data, "SEND OK", "ERROR", 10000);
342   Serial.println(ip_data);
343   Serial.println("*!Send OK!*");
344   i_StateMachine_State = 3;
345   Serial.println("Going from Stage 2 to State 3 - Closeing Socket");
346   }
347   else
348   {
349   Serial.println("Error Sending Data");
350   CloseTcpSocket();
351   i_StateMachine_State = 2;
352   Serial.println("Remaining in State 2 - Resend");
353
354   //if (i_AT_ErrorRetry == AT_ErrorRetryCounter)
355   // {
356   // Serial.println("Shutting Module and Return to Stage 1");
357   // ErrorShutConnection();
358   // }
359   // else
360   // {
361   // Serial.print(("Retying AT command: ") + String(AT_ErrorRetryCounter));
362   // AT_ErrorRetryCounter++;
363   // delay(1000);
364   // }
365   }
366   }
367   }
368   //****************************** State 3 ***********************************//
369   void CloseTcpSocket(){
370   i_endATmillis = millis();
371   //delay(1500);
372   // Closes the socket
373
374   if (sendATcommand2("AT+CIPSTATUS", "STATUS:3", "STATUS:4", 10000) == 1)
375   {
376   Serial.println("Status 3 - Close TCP Socket");
377   CloseSocket();
378   i_StateMachine_State = 1;
379   }
380   else
381   {
382   Serial.println("Status 4 - TCP Socket Closed");
383   i_StateMachine_State = 1;
384   }
385
386
387
388
389   /*
390
391   if (sendATcommand2("AT+CIPCLOSE", "CLOSED", "ERROR", 10000) == 1)
392   {
393   // Waits for status IP STATUS
394   Serial.println("TCP Socket Closed");
395   i_StateMachine_State = 1;
```

```
396   Serial.println("Going from State 3 to State 1");
397   }
398   else
399   {
400   Serial.println("Error Closing the connection");
401   i_StateMachine_State = 3;
402   Serial.println("Staying in State 3 - Close Socket");
403
404   if (i_AT_ErrorRetry == AT_ErrorRetryCounter)
405   {
406   Serial.println("Shutting Module and Return to Stage 1");
407   i_StateMachine_State = 1;
408   }
409   else
410   {
411   Serial.print(("Retying AT command: ") + String(AT_ErrorRetryCounter));
412   AT_ErrorRetryCounter++;
413   delay(1000);
414   }
415   } */
416   i_StateMachineMillis = i_endATmillis - i_startATmillis;
417   b_StateMachine = false;
418   }
419   //************************************************************************//
420   //************************************************************************//
421   void CloseSocket(){
422   if (sendATcommand2("AT+CIPCLOSE", "CLOSED", "ERROR", 10000) == 1)
423   {
424   // Waits for status IP STATUS
425   Serial.println("TCP Socket Closed");
426   //i_StateMachine_State = 1;
427   Serial.println("Going from State 3 to State 1");
428   }
429   else
430   {
431   CloseTcpSocket();
432   }
433   return;
434   }
435   //************************************************************************//
436   //***************************** State Machine ****************************//
437   void StateMachine(){
438   b_StateMachine = true;
439
440   while (b_StateMachine)
441   {
442   if (i_StateMachine_State == 1)
443   {
444   CheckConnection();
445   }
446   if (i_StateMachine_State == 2)
447   {
448   SendData();
449   }
450   if (i_StateMachine_State == 3)
451   {
452   CloseTcpSocket();
453   }
```

```
454   }
455
456   }
457
458   //****************************** Tx Rx Data ***************************//
459   int8_t sendATcommand2(char* ATcommand, char* expected_answer1,
460   char* expected_answer2, unsigned int timeout){
461
462   uint8_t x=0,        answer=0;
463   char response[4096];
464   unsigned long previous;
465
466   memset(response, '\0', 4096);          // Initialize the string
467
468   delay(100);
469
470   while( Serial1.available() > 0) Serial1.read(); // Clean the input buffer
471
472   Serial1.println(ATcommand);// Send the AT command
473   //Serial.println(ATcommand);
474
475   x = 0;
476   previous = millis();
477
478   // this loop waits for the answer
479   do{
480   // if there are data in the UART input buffer, reads it and checks for the asnwer
481   if(Serial1.available() != 0){
482   response[x] = Serial1.read();
483
484   // find and print non printable characters
485   //Serial.println("<<<<<<<<<<< Non Alpha >>>>>>>>>>>>>>>>>");
486   /*char tmp[100];
487   char t = response[x];
488   sprintf(tmp, "%c = %d", t, t);
489   Serial.println(tmp); */
490
491   x++;
492   // check if the desired answer 1 is in the response of the module
493   if (strstr(response, expected_answer1) != NULL)
494   {
495   answer = 1;
496   }
497   // check if the desired answer 2 is in the response of the module
498   else if (strstr(response, expected_answer2) != NULL)
499   {
500   answer = 2;
501   }
502   }
503   //else {
504   //Serial.println("No data on Serial1 buffer");
505   //}
506   }
507   // Waits for the asnwer with time out
508   while((answer == 0) && ((millis() - previous) < timeout));
509
510   Serial.print("*ESP8266-01: ");
511   Serial.println(response);
```

```
512   return answer;
513   }
514
515   //************************************************************************//
516   void GetThingSpeakFieldValues(){
517
518   char ConcatIndex[] = "";
519   sprintf(ConcatIndex, "%ld%s%d",dataIndex, ".", i);
520   Serial.print("****************************** ");
521   Serial.print(ConcatIndex);
522   Serial.println(" ******************************");
523
524   //fieldValue_1 = dataIndex;
525   sprintf(fieldValue_1, "%ld%s%d",dataIndex, ".", i);
526   fieldValue_2 = dataHour;
527   fieldValue_3 = dataMinute;
528   fieldValue_4 = dataSecond;
529   fieldValue_5 = RawDataVolt;
530   fieldValue_6 = RawDataAmp;
531   fieldValue_7 = RawDataTemperature;
532   fieldValue_8 = i_StateMachineMillis;
533   }
534   //************************************************************************//
535   //************************************************************************//
536   void PingSomething(){
537   sprintf(AT_String, "");
538   sprintf(AT_String, "AT+PING=\"%s\"", PingIP);
539   Serial.println(AT_String);
540
541   if (sendATcommand2(AT_String, "OK", "ERROR", 2000) == 1)
542   {
543   Serial.println("Ping was Sussessfull");
544   }
545   else
546   {
547   Serial.println("No Reply");
548   }
549   }
550   //************************************************************************//
551   void WiFiConnect(){
552   sprintf(AT_String, "");
553   sprintf(AT_String, "AT+CWJAP=\"%s\",\"%s\"", ssid, pass);
554   Serial.println(AT_String);
555
556   if (sendATcommand2(AT_String, "WIFI GOT IP", "ERROR", 5000) == 1)
557   {
558   Serial.println("WiFi Connection was Sussessfull");
559   }
560   else
561   {
562   Serial.println("WiFi Connection Error");
563   }
564   }
565   //************************************************************************//
566   // Set ESP to Station Mode
567   void SetEspMode(){
568
569   if (sendATcommand2("AT+CWMODE=1", "OK", "ERROR", 1000) == 1)
```

```
{
Serial.println("ESP8266 Set to Station Mode = 1");
}
else
{
Serial.println("Error setting ESP Mode");
}
}
//*********************************************************************//
boolean  CheckWiFiRouterConnection(){
sprintf(AT_String,  "");
sprintf(AT_String,  "AT+PING=\"%s\"",  RouterIP);
Serial.println(AT_String);

if  (sendATcommand2(AT_String,  "OK",  "ERROR",  2000) ==  1)
{
Serial.println("Ping was Sussessfull");
return  true;
}
else
{
Serial.println("Ping Failed");
return  false;
}
}
//*********************************************************************//
```

Annexure C: GPRS Class 10 Program

```
1   //#include <SPI.h>
2   //#include <SD.h>
3   #include <ModbusRtu.h>
4   //#include <SoftwareSerial.h>
5   //#include <LiquidCrystal.h>
6
7   /********************** Define Modbus Parameters ********************/
8   #define serialBaud 115200                    // 115200 Monitor port
9   #define serialBaud1 9600                     // 115200 GSM Port
10  #define serialBaud2 19200                    // 115200 Modbus Port
11  #define serialBaud3 19200                    // 115200 Modbus Port
12  #define ModbusPortNum 2 // serial port number for Modbus Master
13  #define portSetup SERIAL_8N2 // serial port setup for Modbus Master
14  #define hardwareType 0 // 0 for RS-232 and USB-FTDI or any pin number > 1 for RS-485
15  //#define ssBaud 19200 // SoftwareSerial Bit rate 8N2
16  //#define ssRxPin 6 // SoftwareSerial Rx Pin
17  //#define ssTxPin 7 // SoftwareSerial Tx Pin
18  #define slaveAddress 3 // node id = 0 for master, = 1..247 for slave
19  //#define hardwareType 0 // 0 for RS-232 and USB-
    FTDI or any pin number > 1 for RS-485
20  #define numberOfRegisters 12
21  #define i_AT_ErrorRetry 3 // Number of Retries when Error is Returned
22  #define i_DeadModemRetry 3 // Number of Retries when Dead Modem is Detected
23
24
25
26  uint16_t au16data[numberOfRegisters] = {0,0,0,0,0,0,0,0,0,0,0,0}; // data array for modbus
27
28  Modbus slave(slaveAddress,ModbusPortNum,hardwareType);
29
30  //SoftwareSerial mySerial(ssRxPin, ssTxPin);//Rx-Tx - Create a SoftwareSerial object.
31
32
33  /********************** Declare Data Integers ************************/
34  int newData = 0;
35  int softReset = 0;
36  long int dataIndex = 0;
37  int dataYear = 0;
38  int dataMonth = 0;
39  int dataDay = 0;
40  int dataHour = 0;
41  int dataMinute = 0;
42  int dataSecond = 0;
43  int RawDataTemperature = 0;
44  int RawDataAmp = 0;
45  int RawDataVolt = 0;
46  float dataTemperature = 0;
47  float dataAmp = 0;
48  float dataVolt = 0;
49
50  int AT_ErrorRetryCounter = 1;
51  //int fieldValue_1 = 10;
52  char fieldValue_1[12] = "";
53  int fieldValue_2 = 20;
```

```
54  int fieldValue_3 = 30;
55  int fieldValue_4 = 40;
56  int fieldValue_5 = 50;
57  int fieldValue_6 = 60;
58  int fieldValue_7 = 70;
59  int fieldValue_8 = 0;
60  //int rampValue = 1;
61  int i_StateMachine_State = 1;
62  int i_startATmillis = 0;
63  int i_endATmillis = 0;
64  int i_StateMachineMillis = 0;
65
66  //int inPin = 8;
67
68  String dataDate = "";
69  String dataTime = "";
70  String sdDataString = "";
71
72  bool b_StateMachine = false;
73
74  int i = 0;
75
76  char aux_str[200];
77  char ip_data[1024];
78  char ThingSpeakKey[] = "3JUK5TK3IUUEKRPZ";
79
80
81  /************************ LCD Pin Allocation ***************************/
82
      //LiquidCrystal lcd(27, 26, 25, 24, 23, 22); // initialize the library with the numbers of the interfa
    ce pins
83
84  void setup() {
85  /************************* Serial Port Setup ***************************/
86  Serial.begin(serialBaud);
87  while (!Serial) {
88  ; // wait for serial port to connect. Needed for Leonardo only
89  }
90  /************************* Serial for Modbus Setup ************************/
91  slave.begin( serialBaud2, portSetup ); // begin the ModBus object.
92  //slave.begin( mySerial, portSetup ); //begin the ModBus object.
93
94  /******************************** LCD Setup ********************************/
95  //lcd.begin(16, 2);
      // set up the LCD's number of columns and rows:
96  //lcd.print("Data Logger Ver5.5"); // Print a message to the LCD.
97  //delay(2000);
98  //lcd.clear();
99  /********************************** GSM Setup ***************************/
100 pinMode(9, OUTPUT);
101 Serial1.begin(serialBaud1);
102 Serial.println("Starting...");
103 Serial.println("Connecting to the network...");
104 if (sendATcommand2("AT", "OK", "", 1000) == 2)
105 {
106 Serial.println("GSM module is Powered-Up.");
107 delay(250);
108 }
```

```
else
{
Serial.println("Powering-Up GSM module.");
digitalWrite(9, HIGH);
delay(500);
digitalWrite(9, LOW);
delay(5000);;
}
while( sendATcommand2("AT+CREG?", "+CREG: 0,1", "+CREG: 0,5", 1000)== 0 );



}

/************************ Soft Reset ************************************/
void(* resetFunc) (void) = 0;//declare reset function at address 0
// resetFunc(); //Call Reset
/*********************** End Soft Reset *********************************/


/****************************************************************************/
void loop() {
updateSlaveValues();
newDataAvailable();
//softResetRTU();

}

/****************************************************************************/





/****************************************************************************/
void updateSlaveValues() {
slave.poll( au16data, 12 );

newData = au16data[0];
dataIndex = au16data[1];
dataYear = au16data[2];
dataMonth = au16data[3];
dataDay = au16data[4];
dataHour = au16data[5];
dataMinute = au16data[6];
dataSecond = au16data[7];
RawDataVolt = au16data[8];
RawDataAmp = au16data[9];
RawDataTemperature = au16data[10];
softReset = au16data[11];

//dataVolt = RawDataVolt / 100.0;
//dataAmp = RawDataAmp / 100.0;
//dataTemperature = RawDataTemperature / 100.0;
//
  Serial.print(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>          ");
// Serial.println(String(dataVolt));
```

```
165
166     structureDateTime();
167     buildDataString();
168
169
170
171     }
172     /*********************************************************************/
173     void  newDataAvailable() {
174
175     if(newData  ==  1)
176     {
177     au16data[0] =  0;
178     Serial.println(sdDataString);
179     Serial.println(("Volt = ") +  String(au16data[8]) + ("   Amp = ") +String(au16data[9]) + ("
          Temp = ")
+String(au16data[10]));
180     StateMachine();
181     Serial.println("End of State Machine. Exit Loop. newData = "  +  String(newData));
182     i++;
183     }
184     }
185     /*********************************************************************/
186
187     /*********************************************************************/
188     void  buildDataString() {
189     sdDataString  =  "";
190     sdDataString  +=  newData;
191
192     sdDataString  +=  ",";
193     sdDataString  +=  dataIndex;
194
195     sdDataString  +=  ",";
196     sdDataString  +=  dataDate;
197
198     sdDataString  +=  ",";
199     sdDataString  +=  dataTime;
200
201     sdDataString  +=  ",";
202     sdDataString  +=  "Temp  =  ";
203     sdDataString  +=  ",";
204     sdDataString  +=  String(dataTemperature);
205
206     sdDataString  +=  ",";
207     sdDataString  +=  "Voltage  =  ";
208     sdDataString  +=  ",";
209     sdDataString  +=  String(dataVolt);
210
211     sdDataString  +=  ",";
212     sdDataString  +=  "Amp  =  ";
213     sdDataString  +=  ",";
214     sdDataString  +=  String(dataAmp);
215
216     //Serial.println(sdDataString);
217     }
218     /*********************************************************************/
219     void  structureDateTime(){
220
```

```
221   dataDate ="";
222   dataTime ="";
223
224   dataDate = dataDay;
225   dataDate += "/";
226   dataDate += dataMonth;
227   dataDate += "/";
228   dataDate += dataYear;
229
230   dataTime = dataHour;
231   dataTime += ":";
232   dataTime += dataMinute;
233   dataTime += ":";
234   dataTime += dataSecond;
235
236   //Serial.print(dataDate);
237   //Serial.print("  ");
238   //Serial.println(dataTime);
239   }
240
241   /*********************************************************************/
242   /*void softResetRTU()
243   {
244   if(softReset == 1)
245   {
246   Serial.println("Soft Reset");
247   delay(500);
248   resetFunc();
249   }
250   }
251   /*********************************************************************/
252   //*********************************************************************//
253   //****************************** State 1 ******************************//
254   void SelectsSingleConnectionMode(){
255   i_startATmillis = millis();
256
257   // Selects Single-connection mode
258   if (sendATcommand2("AT+CIPMUX=0", "OK", "ERROR", 1000) == 1)
259   {
260   // Waits for status IP INITIAL
261   while(sendATcommand2("AT+CIPSTATUS", "INITIAL", "", 500) == 0 );
262   //delay(5000);
263   i_StateMachine_State = 2;
264   Serial.println("Going from State 1 to State 2");
265   AT_ErrorRetryCounter = 1;
266   }
267   else
268   {
269   Serial.println("Error setting the single connection");
270   i_StateMachine_State = 1;
271   Serial.println("Staying in State 1");
272   //sendATcommand2("AT+CIPSHUT", "OK", "ERROR", 10000);
273   if (i_AT_ErrorRetry == AT_ErrorRetryCounter)
274   {
275   Serial.println("Shutting Module and Return to Stage 1");
276   ErrorShutConnection();
277   }
278   else
```

```
{
Serial.print(("Retying AT command: ") + String(AT_ErrorRetryCounter));
AT_ErrorRetryCounter++;
delay(1000);
}
}
}
//****************************** State 2 ******************************//
void SetsApnUserNamePassword(){
// Sets the APN, user name and password

  if (sendATcommand2("AT+CSTT=\"internet\",\"\",\"\"", "OK","ERROR", 30000) == 1) //
MTN - APN

  //if (sendATcommand2("AT+CSTT=\"afrihost\",\"\",\"\"", "OK","ERROR", 30000) == 1)
  // Afrihost - APN
{
// Waits for status IP START
while(sendATcommand2("AT+CIPSTATUS", "START", "", 500) == 0 );
//delay(5000);
i_StateMachine_State = 3;
Serial.println("Going from State 2 to State 3");
AT_ErrorRetryCounter = 1;
}
else
{
Serial.println("Error setting the APN");
i_StateMachine_State = 2;
Serial.println("Staying in State 2");

if (i_AT_ErrorRetry == AT_ErrorRetryCounter)
{
Serial.println("Shutting Module and Return to Stage 1");
ErrorShutConnection();
}
else
{
Serial.print(("Retying AT command: ") + String(AT_ErrorRetryCounter));
AT_ErrorRetryCounter++;
delay(1000);
}
}
}

//****************************** State 3 ******************************//
void BringUpWirelessConnection(){
// Brings Up Wireless Connection
if (sendATcommand2("AT+CIICR", "OK", "ERROR", 30000) == 1)
{
// Waits for status IP GPRSACT
while(sendATcommand2("AT+CIPSTATUS", "GPRSACT", "", 500) == 0 );
//delay(5000);
i_StateMachine_State = 4;
Serial.println("Going from State 3 to State 4");
AT_ErrorRetryCounter = 1;
}
else
{
```

```
333   Serial.println("Error bring up wireless connection");
334   i_StateMachine_State = 3;
335   Serial.println("Staying in State 3");
336
337   if (i_AT_ErrorRetry == AT_ErrorRetryCounter)
338   {
339   Serial.println("Shutting Module and Return to Stage 1");
340   ErrorShutConnection();
341   }
342   else
343   {
344   Serial.print(("Retying AT command: ") + String(AT_ErrorRetryCounter));
345   AT_ErrorRetryCounter++;
346   delay(1000);
347   }
348   }
349
350   }
351   //****************************** State 4 ******************************//
352   void GetsLocalIpAddress(){
353   // Gets Local IP Address
354   if (sendATcommand2("AT+CIFSR", ".", "ERROR", 10000) == 1)
355   {
356   // Waits for status IP STATUS
357   while(sendATcommand2("AT+CIPSTATUS", "IP STATUS", "", 500) == 0 );
358   //delay(5000);
359   Serial.println("Openning TCP");
360   i_StateMachine_State = 5;
361   Serial.println("Going from State 4 to State 5");
362   AT_ErrorRetryCounter = 1;
363   }
364   else
365   {
366   Serial.println("Error getting the IP address");
367   i_StateMachine_State = 4;
368   Serial.println("Staying in State 4");
369
370   if (i_AT_ErrorRetry == AT_ErrorRetryCounter)
371   {
372   Serial.println("Shutting Module and Return to Stage 1");
373   ErrorShutConnection();
374   }
375   else
376   {
377   Serial.print(("Retying AT command: ") + String(AT_ErrorRetryCounter));
378   AT_ErrorRetryCounter++;
379   delay(1000);
380   }
381   }
382   }
383   //****************************** State ******************************//
384   void SendData(){
385   // Opens a TCP socket
386   if (sendATcommand2("AT+CIPSTART=\"TCP\",\"api.thingspeak.com\",\"80\"",
      "CONNECT OK", "ERROR", 30000) == 1)
387   //if (sendATcommand2("AT+CIPSTART=\"TCP\",\"api.thingspeak.com\",\"80\"",
      "OK ALREADY", "FAIL ERROR", 30000) == 1)
388   {
```

```
389   Serial.println("Connected");
390   Serial.println("Going from State 5 to State 6");
391   GetThingSpeakFieldValues();
392   sprintf(ip_data, "GET/update?api_key=%s&field1=%s&field2=%d&field3=%d&
      field4=%d&field5=%d&field6=%d&field7=%d&field8=%dHTTP
      /1.1\r\nHost: api.thingspeak.com\r\nConnection: keep-alive\r\n\r\n",
393   ThingSpeakKey,
394   fieldValue_1,
395   fieldValue_2,
396   fieldValue_3,
397   fieldValue_4,
398   fieldValue_5,
399   fieldValue_6,
400   fieldValue_7,
401   fieldValue_8);
402   //Serial.println(ip_data);
403
404   sprintf(aux_str,"AT+CIPSEND=%d", strlen(ip_data));
405   if (sendATcommand2(aux_str, ">", "ERROR", 10000) == 1)
406   {
407   sendATcommand2(ip_data, "SEND OK", "ERROR", 10000);
408   Serial.println("*!Send OK!*");
409   i_StateMachine_State = 6;
410   Serial.println("Going from Stage 5 to State 6 - Closeing Socket");
411   AT_ErrorRetryCounter = 1;
412   }
413   else
414   {
415   Serial.println("Error Sending Data");
416   i_StateMachine_State = 5;
417   Serial.println("Staying in State 5 - Resend");
418
419   //if (i_AT_ErrorRetry == AT_ErrorRetryCounter)
420   // {
421   // Serial.println("Shutting Module and Return to Stage 1");
422   // ErrorShutConnection();
423   // }
424   // else
425   // {
426   // Serial.print(("Retying AT command: ") + String(AT_ErrorRetryCounter));
427   // AT_ErrorRetryCounter++;
428   // delay(1000);
429   // }
430   }
431   }
432   else
433   {
434   Serial.println("Error connecting to URL");
435   i_StateMachine_State = 5;
436   Serial.println("Staying in State 5");
437
438   if (i_AT_ErrorRetry == AT_ErrorRetryCounter)
439   {
440   Serial.println("Shutting Module and Return to Stage 1");
441   ErrorShutConnection();
442   }
443   else
444   {
```

```
445   Serial.print(("Retying AT command: ") + String(AT_ErrorRetryCounter));
446   AT_ErrorRetryCounter++;
447   delay(1000);
448   }
449   }
450   }
451   //****************************** State 6 ******************************//
452   void CloseTcpSocket(){
453   i_endATmillis = millis();
454
455   // Closes the socket
456   if (sendATcommand2("AT+CIPCLOSE", "CLOSE OK", "ERROR", 10000) == 1)
457   {
458   // Waits for status IP STATUS
459   Serial.println("IP Socket Closed");
460   i_StateMachine_State = 7;
461   Serial.println("Going from State 6 to State 7");
462   AT_ErrorRetryCounter = 1;
463   //Serial.println("Final Stage - State6. Closing TCP Socket");
464   }
465   else
466   {
467   Serial.println("Error openning the connection");
468   i_StateMachine_State = 6;
469   Serial.println("Staying in State 6 - Close Socket");
470
471   if (i_AT_ErrorRetry == AT_ErrorRetryCounter)
472   {
473   Serial.println("Shutting Module and Return to Stage 1");
474   ErrorShutConnection();
475   }
476   else
477   {
478   Serial.print(("Retying AT command: ") + String(AT_ErrorRetryCounter));
479   AT_ErrorRetryCounter++;
480   delay(1000);
481   }
482   }
483   }
484   //****************************** State 7 ******************************//
485   void ShuttingConnection(){
486   // Shutting Down IP Connection
487   if (sendATcommand2("AT+CIPSHUT", "OK", "ERROR", 10000) == 1)
488   {
489   // Waits for status IP STATUS
490   Serial.println("IP Connection Shut");
491   i_StateMachine_State = 1;
492   Serial.println("Final Stage - State7. Closing Shut Connection");
493   Serial.println("Going from State 7 to State 1");
494   }
495   else
496   {
497   Serial.println("Error closing the IP connection");
498   i_StateMachine_State = 7;
499   Serial.println("Staying in State 7");
500   }
501
502   //Serial.println(("Number of Records >>>>>>>>>>>>>>>>>>>> ") + String(rampValue));
```

```
503   //rampValue++;
504
505   //i_endATmillis = millis();
506   i_StateMachineMillis = i_endATmillis - i_startATmillis;
507   Serial.println(("AT Time To Transmit = >>>>>>>>>>>>>> ") + String(i_StateMachineMillis));
508   b_StateMachine = false;
509   //fieldValue_2 = i_StateMachineMillis //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
510   //delay(30000);                    //290000 = 5min
511
512   }
513   //************************ Error Shut Connection ************************//
514   void ErrorShutConnection(){
515   // Shutting Down IP Connection
516   if (sendATcommand2("AT+CIPSHUT", "OK", "ERROR", 10000) == 1)
517   {
518   // Waits for status IP STATUS
519   Serial.println("IP Connection Shut");
520   i_StateMachine_State = 1;
521   Serial.println("Connection Shut. Error Retry Count Exhausted. Going to State 1");
522   AT_ErrorRetryCounter = 1;
523   }
524   else
525   {
526   Serial.println("Error closing the IP connection");
527   ErrorShutConnection();
528   delay(2000);
529   Serial.println("Retrying ..... ");
530   }
531   }
532   //************************ State Machine ****************************//
533   void StateMachine(){
534   b_StateMachine = true;
535
536   while (b_StateMachine)
537   {
538   if (i_StateMachine_State == 1)
539   {
540   SelectsSingleConnectionMode();
541   }
542   if (i_StateMachine_State == 2)
543   {
544   SetsApnUserNamePassword();
545   }
546   if (i_StateMachine_State == 3)
547   {
548   BringUpWirelessConnection();
549   }
550   if (i_StateMachine_State == 4)
551   {
552   GetsLocalIpAddress();
553   }
554   if (i_StateMachine_State == 5)
555   {
556   SendData();
557   }
558   if (i_StateMachine_State == 6)
559   {
560   CloseTcpSocket();
```

```
561   }
562   if (i_StateMachine_State == 7)
563   {
564   ShuttingConnection();
565   }
566
567   }
568
569   }
570
571   //***************** Tx Rx Data ***************************//
572   int8_t sendATcommand2(char* ATcommand, char* expected_answer1,
573   char* expected_answer2, unsigned int timeout){
574
575   uint8_t x=0,                    answer=0;
576   char response[4096];
577   unsigned long previous;
578   int DeadModemCounter = 0;
579
580   memset(response, '\0', 4096);            // Initialize the string
581
582   delay(100);
583
584   while( Serial1.available() > 0) Serial1.read();   // Clean the input buffer
585
586   Serial1.println(ATcommand);              // Send the AT command
587
588   x = 0;
589   previous = millis();
590
591   // this loop waits for the answer
592   do{
593   // if there are data in the UART input buffer, reads it and checks for the asnwer
594   if(Serial1.available() != 0){
595   response[x] = Serial1.read();
596
597   // find and print non printable characters
598   /* Serial.println("<<<<<<<<<< Non Alpha >>>>>>>>>>>>>>>>>>");
599   char tmp[100];
600   char t = response[x];
601   sprintf(tmp, "%c = %d", t, t);
602   Serial.println(tmp); */
603
604   int i = response[x];
605   x++;
606
607   //Serial.print("Responce = ");
608   //Serial.println(response);
609   // check if the desired answer 1 is in the response of the module
610   if (strstr(response, expected_answer1) != NULL)
      //compare 2 strings with each other - "response" and "expected_answer1"
611   //if (strstr(expected_answer1, response) != NULL)
612   {
613   answer = 1;
614   }
615   // check if the desired answer 2 is in the response of the module
616   else if (strstr(response, expected_answer2) != NULL)
617   //else if (strstr(expected_answer2, response) != NULL)
```

```
618   {
619   answer = 2;
620   }
621   // cheack if the answer is nagative. If true, modem Powered Off. Enable modem
622   else if (i < 0)
623   {
624   DeadModemCounter++;
625   Serial.print("Dead Modem Count: ");
626   Serial.println(DeadModemCounter);
627   if (DeadModemCounter == i_DeadModemRetry)
628   {
629   EnableModem();
630   }
631   }
632   }
633   }
634   // Waits for the asnwer with time out
635   while((answer == 0) && ((millis() - previous) < timeout));
636
637   Serial.print("*SIM900 response: ");
638   Serial.println(response);
639   return answer;
640   }
641
642   //*************************************************************//
643   void EnableModem(){
644   Serial.println("Enable Sent to Modem");
645   // delay(500);
646   digitalWrite(9, HIGH);
647   delay(500);
648   digitalWrite(9, LOW);
649   delay(5000);
650   }
651   //*************************************************************//
652   void GetThingSpeakFieldValues(){
653
654   //fieldValue_1 = dataIndex;
655   sprintf(fieldValue_1, "%ld%s%d",dataIndex, ".", i);
656   fieldValue_2 = dataHour;
657   fieldValue_3 = dataMinute;
658   fieldValue_4 = dataSecond;
659   fieldValue_5 = RawDataVolt;
660   fieldValue_6 = RawDataAmp;
661   fieldValue_7 = RawDataTemperature;
662   fieldValue_8 = i_StateMachineMillis;
663   }
664   //*************************************************************//
665
666
667
668
```

Annexure D: XBee Unit Program

```
#include <SPI.h>
#include <SD.h>
#include <ModbusRtu.h>
#include <SoftwareSerial.h>
//#include <LiquidCrystal.h>

/********************** Define Modbus Parameters *********************/
#define serialBaud 115200                    // 115200
#define serialBaud1 19200                    // 115200
#define portNum 1 // serial port number for Modbus Master
#define portSetup SERIAL_8N2 // serial port setup for Modbus Master
#define hardwareType 0 // 0 for RS-232 and USB-FTDI or any pin number > 1 for RS-485
//#define ssBaud 19200 // SoftwareSerial Bit rate 8N2
//#define ssRxPin 6 // SoftwareSerial Rx Pin
//#define ssTxPin 7 // SoftwareSerial Tx Pin
//#define pollTimeout 1000 // 1000
//#define scanRate 100 // the scan rate (100 min for SoftwareSerial)
//#define masterAddress 0 // node id = 0 for master, = 1..247 for slave
#define slaveAddress 1 // node id = 0 for master, = 1..247 for slave
//#define hardwareType 0 // 0 for RS-232 and USB-FTDI or any pin number > 1 for RS-485
//#define numberOfRegisters 5

uint16_t au16data[12] = {0,0,0,0,0,0,0,0,0,0,0,0}; // data array for modbus

Modbus slave(slaveAddress,portNum,hardwareType);

//SoftwareSerial mySerial(ssRxPin, ssTxPin);//Rx-Tx - Create a SoftwareSerial object.


/********************** Declare Data Integers ************************/
int newData = 0;
int softReset = 0;
long int dataIndex = 0;
int dataYear = 0;
int dataMonth = 0;
int dataDay = 0;
int dataHour = 0;
int dataMinute = 0;
int dataSecond = 0;
float dataTemperature = 0;
float dataAmp = 0;
float dataVolt = 0;

int inPin = 8;

String dataDate = "";
String dataTime = "";
String sdDataString = "";


int i = 0;

```

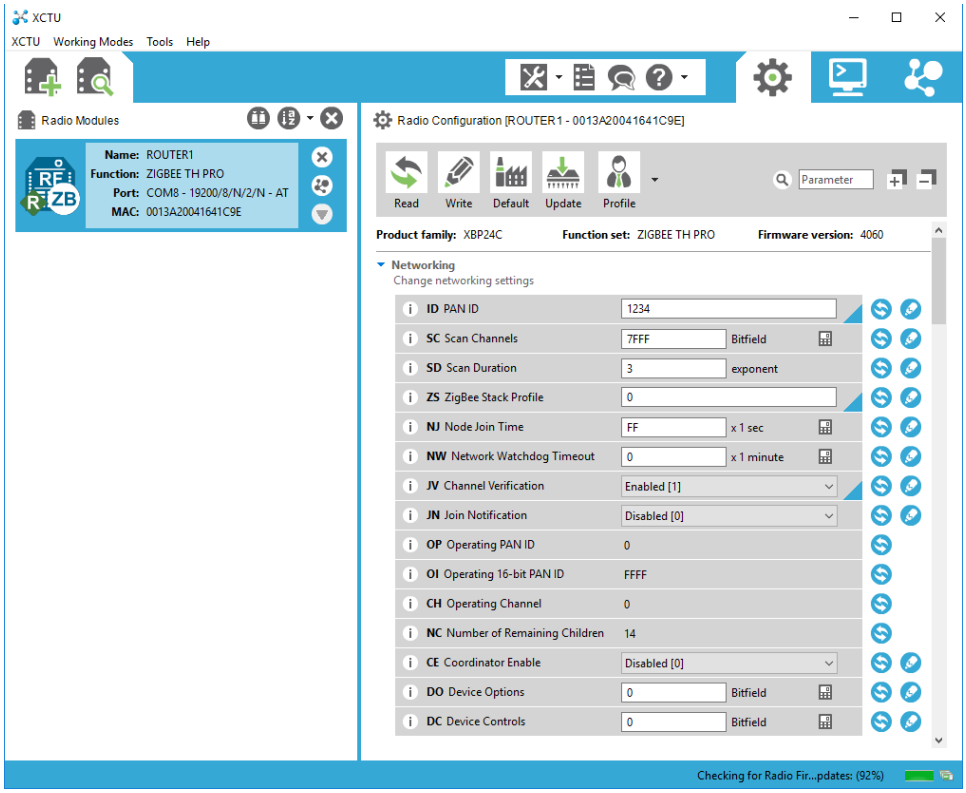
```
53
54  /********** SD Card Pin Allocation and File Assignment ************/
55  const int chipSelect = 4;
      //10 Werk op UNO en 53 werk op die Mega
56  //File dataFile;
57  File mainLogFile;
58  //String myReadString;
59
60  /********************** LCD Pin Allocation **************************/
61
    //LiquidCrystal lcd(27, 26, 25, 24, 23, 22);  // initialize the library with the numbers of the i
    nterface pins
62
63
64
65
66
67
68
69  void setup() {
70  /************************ Serial Port Setup ***************************/
71  Serial.begin(serialBaud);
72  while (!Serial) {
73  ;  // wait for serial port to connect. Needed for Leonardo only
74  }
75  /************** Soft Serial for Modbus Setup ************************/
76  slave.begin( serialBaud1, portSetup ); // begin the ModBus object.
77
78
79
80
81
82
83
84
85
86
87  //lcd.begin(16, 2);
      // set up the LCD's number of columns and rows:
88  //lcd.print("Data Logger Ver5.5");
      // Print a message to the LCD.
89  //delay(2000);
90  //lcd.clear();
91
92  /************************ SD Card Setup Start ************************/
93  Serial.print("Initializing SD card...");
94  // make sure that the default chip select pin is set to
95  // output, even if you don't use it:
96  pinMode(chipSelect, OUTPUT);
97
98  // see if the card is present and can be initialized:
99  if (!SD.begin(chipSelect)) {
100 Serial.println("Card failed, or not present");
101 // don't do anything more:
102 return;  //while (1) ;
103 }
104 Serial.println("card initialized.");
105
```

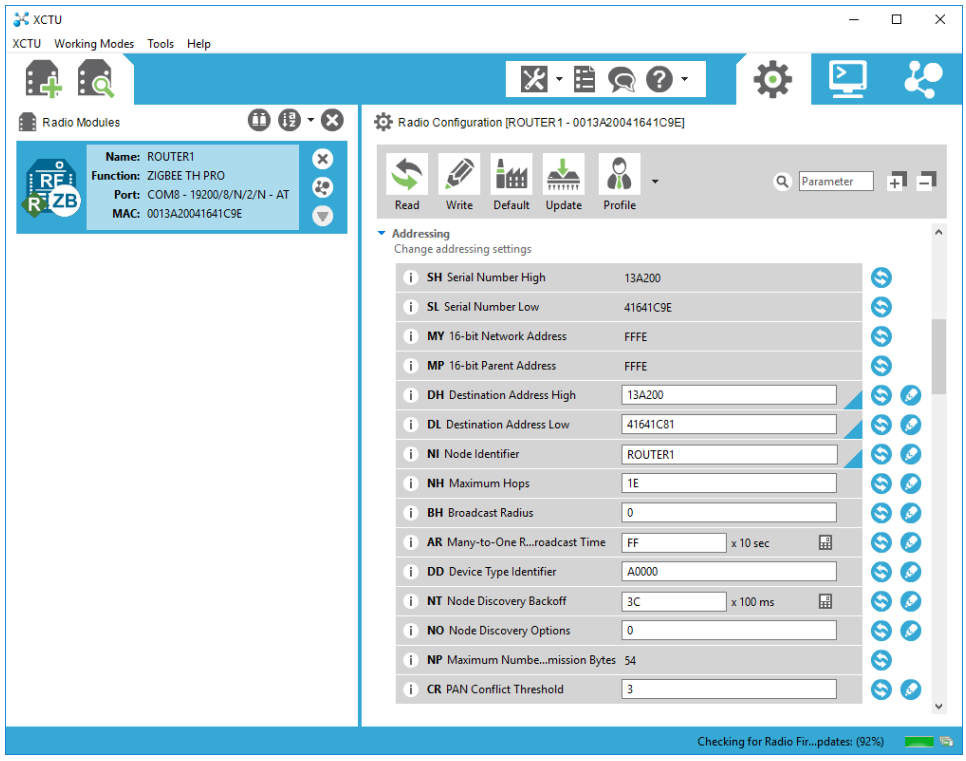
```
106
107 /********* Create New LOG file with a restart method ****************/
108 // char filename[] = "DATA0000.CSV";
109 // for (uint8_t i = 0; i < 10000; i++) {
110 // filename[6] = i/10 + '0';
111 // filename[7] = i%10 + '0';
112 // if (! SD.exists(filename)) {
113 // // only open a new file if it doesn't exist
114 // dataFile = SD.open(filename, FILE_WRITE);
115 // break; // leave the loop!
116 // }
117 // }
118
119 // Serial.print("Logging to: ");
120 // Serial.println(filename);
121
122 //filename = SD.open(filename, FILE_WRITE);
123
124
125 /********* Append LOG file with a restart method *****************/
126 // Open up the file we're going to log to!
127 mainLogFile = SD.open("LogSD.CSV", FILE_WRITE);
128 if (! mainLogFile) {
129 Serial.println("error opening MainLog.CSV");
130 // Wait forever since we cant write data
131 return; //while (1) ;
132 }
133
134 mainLogFile = SD.open("LogSD.CSV", FILE_WRITE);
135
136 Serial.print("Logging to: ");
137 Serial.println("LogSD.CSV");
138
139
140
141 pinMode(inPin, INPUT);
142 }
143
144
145 /************************ Soft Reset *******************************/
146 void(* resetFunc) (void) = 0;//declare reset function at address 0
147 // resetFunc(); //Call Reset
148 /*********************** End Soft Reset ****************************/
149
150
151 /*********************************************************************/
152 void loop() {
153 updateSlaveValues();
154 newDataAvailable();
155 softResetRTU();
156
157 }
158
159 /*********************************************************************/
160
161
162 /*********************************************************************/
163 void updateSlaveValues() {
```

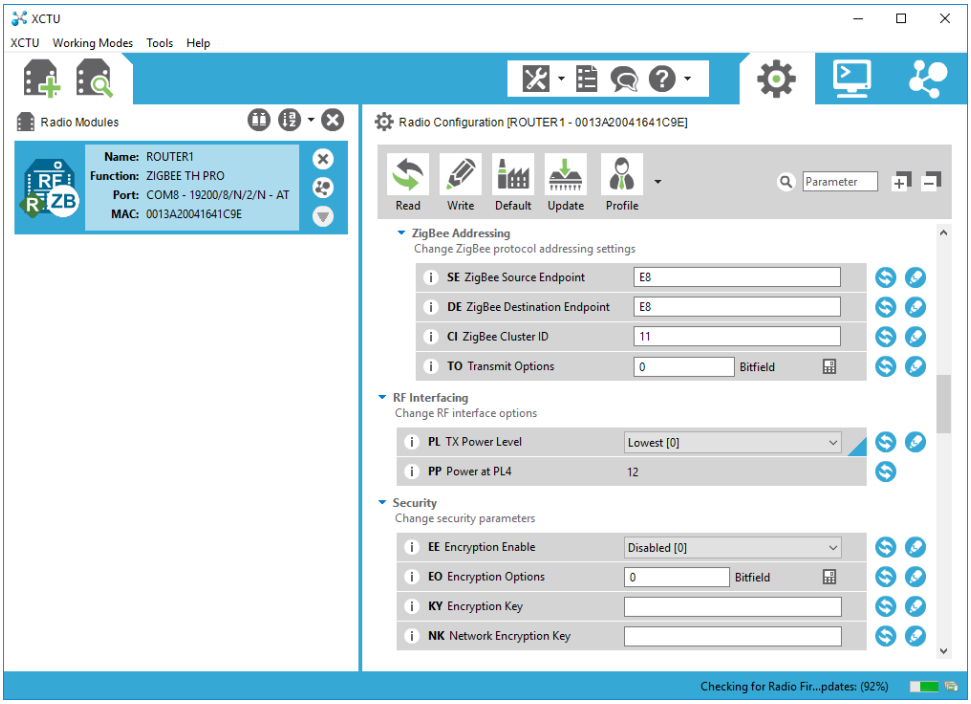
```
164   slave.poll( au16data, 12 );
165
166   newData = au16data[0];
167   dataIndex = au16data[1];
168   dataYear = au16data[2];
169   dataMonth = au16data[3];
170   dataDay = au16data[4];
171   dataHour = au16data[5];
172   dataMinute = au16data[6];
173   dataSecond = au16data[7];
174   dataVolt = au16data[8];
175   dataAmp = au16data[9];
176   dataTemperature = au16data[10];
177   softReset = au16data[11];
178
179   dataVolt = dataVolt / 100;
180   dataAmp = dataAmp / 100;
181   dataTemperature = dataTemperature / 100;
182
183   structureDateTime();
184   buildDataString();
185
186   }
187   /*********************************************************************/
188   void newDataAvailable() {
189
190   if(newData == 1)
191   {
192   //structureDateTime();
193   writeToSD();
194   au16data[0] = 0;
195   Serial.println(sdDataString);
196   i++;
197   }
198   }
199   /*********************************************************************/
200   void buildDataString() {
201   sdDataString = "";
202   sdDataString += newData;
203
204   sdDataString += ",";
205   sdDataString += dataIndex;
206
207   sdDataString += ",";
208   sdDataString += i;
209
210   sdDataString += ",";
211   sdDataString += dataDate;
212
213   sdDataString += ",";
214   sdDataString += dataTime;
215
216   sdDataString += ",";
217   sdDataString += "Temp = ";
218   sdDataString += ",";
219   sdDataString += dataTemperature;
220
221   sdDataString += ",";
```

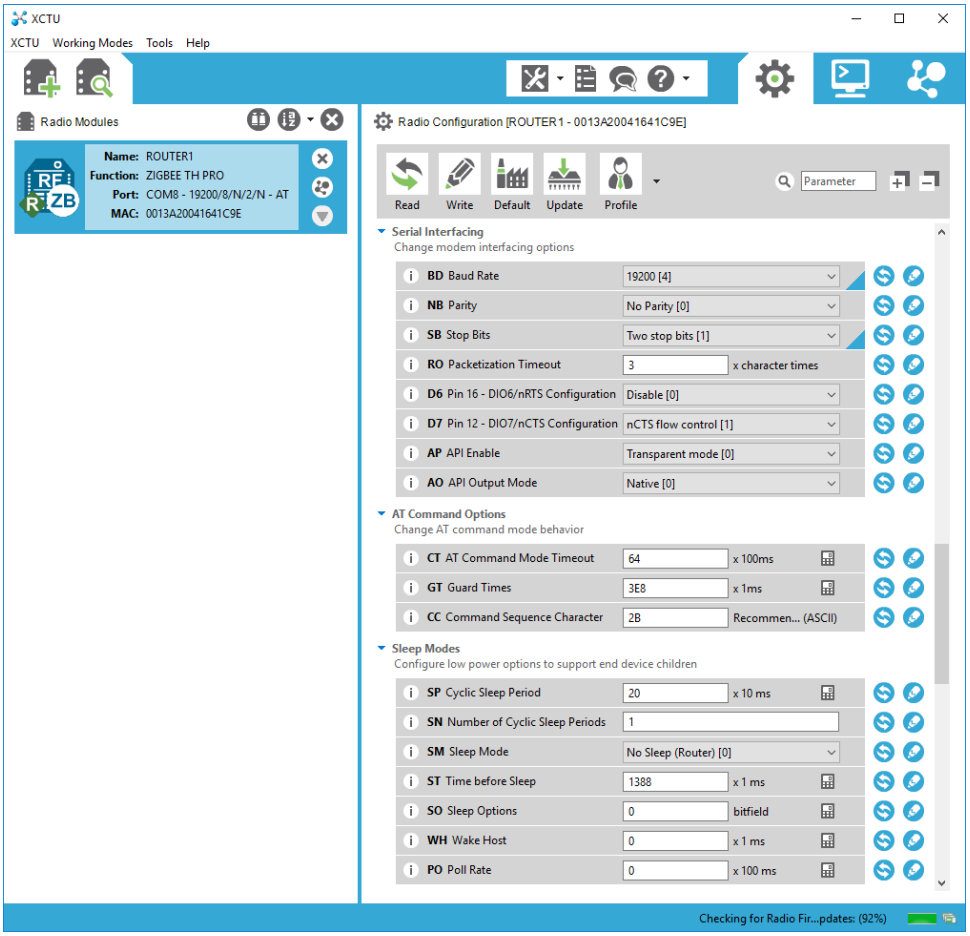
```
222   sdDataString += "Voltage = ";
223   sdDataString += ",";
224   sdDataString += dataVolt;
225
226   sdDataString += ",";
227   sdDataString += "Amp = ";
228   sdDataString += ",";
229   sdDataString += dataAmp;
230
231   //Serial.println(sdDataString);
232   }
233
      /*************************************************************************
      **/
234   void structureDateTime(){
235
236   dataDate ="";
237   dataTime ="";
238
239   dataDate = dataDay;
240   dataDate += "/";
241   dataDate += dataMonth;
242   dataDate += "/";
243   dataDate += dataYear;
244
245   dataTime = dataHour;
246   dataTime += ":";
247   dataTime += dataMinute;
248   dataTime += ":";
249   dataTime += dataSecond;
250
251   //Serial.print(dataDate);
252   //Serial.print("  ");
253   //Serial.println(dataTime);
254   }
255   /*************************************************************************/
256   void writeToSD(){
257
258   //Serial.println(sdDataString);
259
260   //dataFile.println(sdDataString);
261   mainLogFile.println(sdDataString);
262
263   //dataFile.flush();
264   mainLogFile.flush();
265
266   }
267   /*************************************************************************/
268   void softResetRTU()
269   {
270   if(softReset == 1)
271   {
272   Serial.println("Soft Reset");
273   delay(500);
274   resetFunc();
275   }
276   }
277   /*************************************************************************/
```

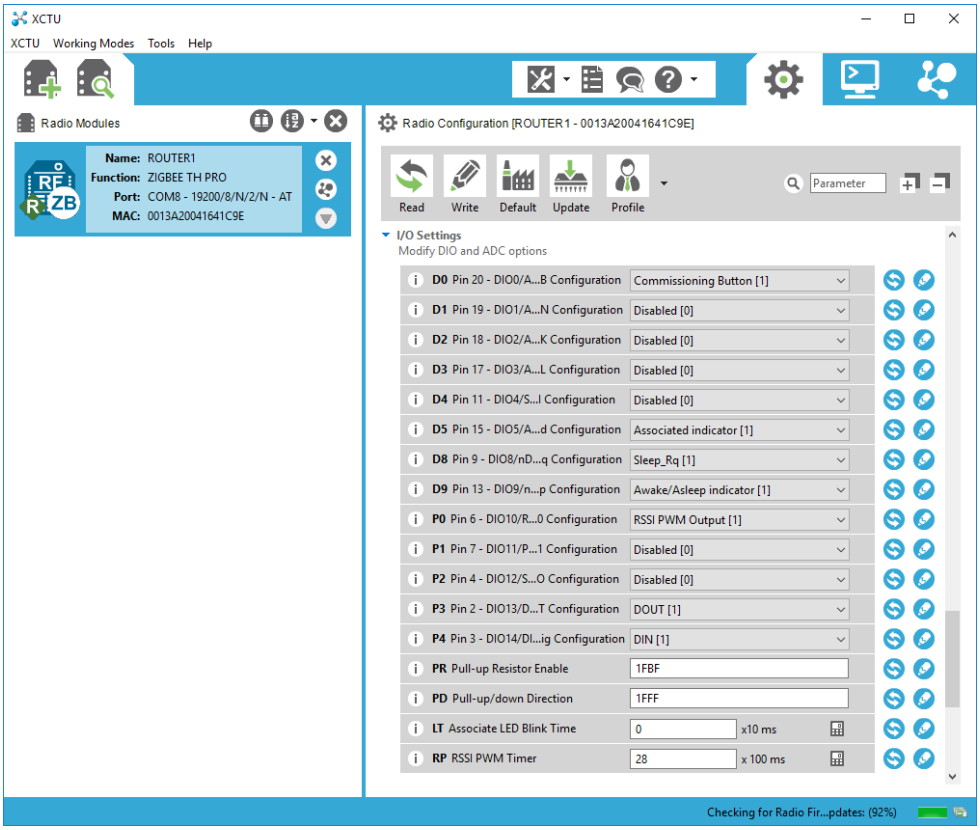
Annexure E: XBee Configuration & Test Utility Software (XCTU)

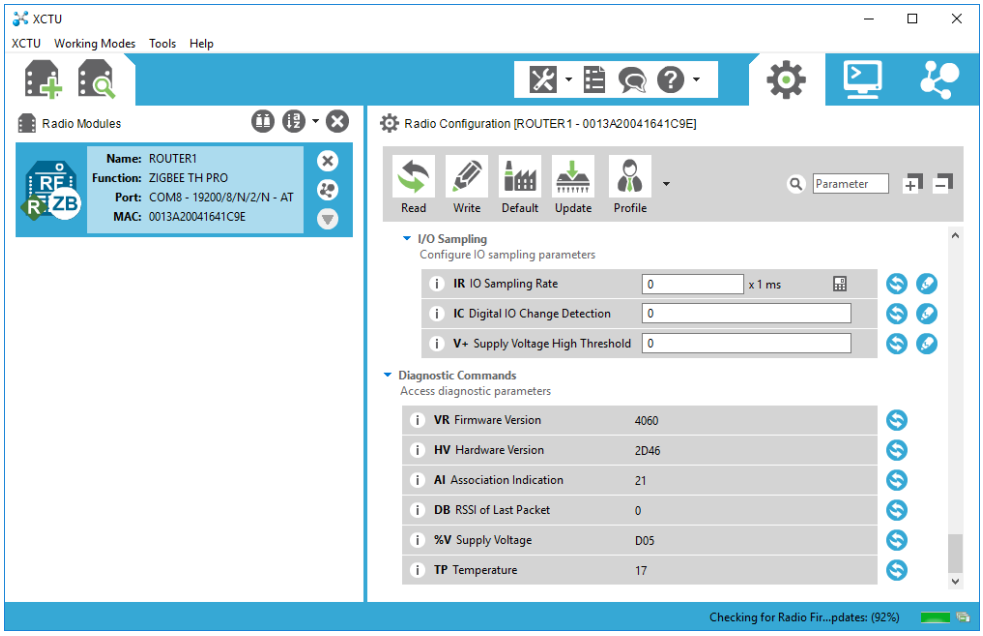












Annexure F: Paper Presented at SAUPEC 2019 Conference

Propagation Delays and Data Integrity of Cellular and WiFi Networks from IOT devices to cloud storage

William Benjamin Van Der Merwe
Department of Electrical, Electronic and Computer Engineering
Central University of Technology
Bloemfontein, South Africa
vdmerwwb@eskom.co.za

Pierre E Hertzog
Department of Electrical, Electronic and Computer Engineering
Central University of Technology
Bloemfontein, South Africa
phertzog@cut.ac.za

Arthur J Swart
Department of Electrical, Electronic and Computer Engineering
Central University of Technology
Bloemfontein, South Africa
aswart@cut.ac.za

***Abstract*— Transmitting data for cloud storage via the internet with wireless technology is becoming more and more important for researchers, hobbyists and commercial applications. High speed internet connections, such as fiber optic and LTE connections enable users to transmit data to cloud storage in a very fast manner. Taking this into account the question now arises which technology is the more dependable, efficient and cost effective method for sensors to transmit their data via the internet to a cloud storage environment.**

The aim of this paper is to investigate the propagation delay and data integrity of transmitted data from cellular and WiFi networks to cloud storage. The Main or Sampling unit continuously evaluates the analogue inputs. It will then send the sampled data every so often to the technologies under test. The slave units will then send the sampled data and index number received from the Main unit to cloud storage via their respective communication technologies. The main unit will also record the sampled values, index number and time and date stamp from the GPS to an onboard logger to keep track of the sampled values. This indexing system will then be used to evaluate data integrity and propagation delay. By comparing the sampler units logged data (index, data and time and date stamp information) to that of the slave units cloud data a clear picture of data integrity and propagation delay can be concluded for the technologies under test. The results indicate that WiFi is a quicker option to use compared to cellular, but might be more expensive if a small amount of data is transmitted. The outcome of this research may help researchers, hobbyists and commercial users to make a better informed decision about the technology they wish to use for their particular environment.

***Keywords*— Wireless Fidelity, Long Term Evolution communications, GPRS CLASS 10, Indexing system**

I. INTRODUCTION

"The world is being re-shaped by the convergence of social, mobile, cloud, big data, community and other powerful forces. The combination of these technologies unlocks an incredible opportunity to connect everything together in a new way and is dramatically transforming the way we live and work."– 2014 [1], Marc Benioff. The ability to send sensor data to a cloud server for storage or connect directly to any sensor or cloud server from anywhere in the world is becoming more and more important for researchers, hobbyists and even commercial use. However a cost comparison must be done by the user where the cost of cellular communications must be compared to that of Asymmetric Digital Subscriber Line (ADSL) or Fiber optic, taking into account the amount of data that needs to be sent as well as the geographical area [2]. The opportunity to connect everything with everything has grown from strength to strength in the last couple of years, taking into account the amount of sensors available today and the rapid development of new ones [3]. Thus the internet, in conjunction with the Internet of Things (IoT), has enabled one to register many devices on cloud services via the internet, and pass information to a cloud server [4]. The essence of these IoT devices is the ability to create a path for data to flow from strategic locations to a cloud server. Cloud storage provides researchers with the ability to access accumulated data from many platforms, anywhere in the world, with the added advantage that the data is always backed up [5].

Researchers use electronic transducers and sensors to measure the results of their experiments and then transmit the data to be stored in an internet cloud database. Thus, it could be asked, which is the most reliable, expeditious and cost effective method to transmit sensor data through the internet to a cloud storage platform. The aim of this paper is to investigate the propagation delay and data integrity of transmitted data from cellular and WiFi networks to cloud storage. The approach aims to use two different types of known technologies (cellular data communication and WiFi).

This paper firstly covers a review of IoT, IoT devices and cloud storage via ThingSpeak. Secondly, the practical setup is discussed followed by the results and conclusions.

II. IOT

The evolution of a new era of the IoT is upon us [6]. It offers mankind the ability to measure, gather and analyse environmental indicators instantaneously. The ability for devices or sensors to communicate seamlessly with a cloud storage network creates the IoT [7]. For example, consider wearables, like smartphones with installed sensors, which collect data and information about the user. The IoT provides the opportunity for these sensors, home appliances and software to share and communicate their information to each other via the internet [8]. Taking this into account, one must remember that large amounts of private information may also be vulnerable to cyber-crime, giving rise to security issues. However, connecting IoT devices directly to the internet is growing at an alarming pace. "What is missing is a well-architected system that extends the functionality of the cloud and provides seamless interplay

among the heterogeneous components in the IoT" [9]. Added to this is the need to get data to the cloud network as quick as possible and ensured data integrity, which has been defined as the assurance that the data received is an exact replica of what was originally transmitted by an authorized entity [10].

A. ARDUINO MEGA 2560

The Arduino Mega 2560 microcontroller was born after the success of the Arduino Uno in September 2010 [11]. The platform of the Arduino series is open source and widely used for electronic projects. It is inexpensive and runs on any computer operating system. Programing is done in C++ that supports all levels of programing skills. Some externally added sensors or modules require additional libraries. These libraries add different functions and calculations to the programming environment, which makes it easier to use[12].

B. ESP8366

A low-cost WiFi microchip used in this study is the ESP8266 microcontroller and supports the full TCP/IP stack. The chip first came to attention in August 2014 with the ESP-01 produced by a third-party manufacturer, Ai-Thinker. The module makes it possible for microcontrollers to connect serially to a WiFi network using Hayes commands[13]. The Arduino Mega microcontroller can connect to the ESP8266 using communication port 1 to receive data from the master unit, and communication port 2 to transmit received data via WiFi to ThingSpeak cloud server for data storage[14].

C. GSM GPRS Class 10 SIM900

Machine-to-Machine communication connections are made possible with wireless modules. They transmit and decode data over a cellular network. There are two basic wireless networks for which modules and machine-to-machine solutions are developed – GSM and CDMA[15]. The Arduino Mega microcontroller can connect to the SIM900 GPRS CLASS 10 module using communication port 1 to receive data from the master unit, and communication port 2 to transmit received data via GPRS CLASS 10 to ThingSpeak cloud server for data storage[14].

D. THINGSPEAK

The developers of ThingSpeak created this platform as an open source IoT application to store and retrieve data for things using it. ThingSpeak provides the facility for sensors to store their data from logging applications. The web site was originally launched in 2010 as a service for IoT applications[16]. The core element of ThingSpeak is the 'ThingSpeak Channel'. A channel stores the data that is send to ThingSpeak with the following basic elements:

- 8 fields for storing data of any type - These can be used to store the data from a sensor or from an embedded device.
- 3 location fields - Can be used to store the latitude, longitude and the elevation. These are very useful for tracking a moving device.
- 1 status field - A short message to describe the data stored in the channel.

III. PROJECT DESIGN

The aim of this paper is to investigate the propagation delay and data integrity of transmitted data from cellular and WiFi networks to cloud storage.

The research unit is designed and built with standard off the shelf electronic equipment and an Arduino Mega 2560 microcontroller equipped with various shields, as per the technology used. Cyclic measurements of the output voltage, current and temperature of a PV panel will be taken. See Fig. 1 for the block diagram of the practical setup.

The sampling unit is used to evaluate the analogue inputs periodically, sending the sampled data to the two technologies under test. The sampling unit also records these sampled values to an onboard logger to keep track of the values by means of an indexing system. The index system is used to evaluate data integrity. The slave units will then send the sampled data and index to cloud storage via their respective communication technologies. The cloud stored data can then be compared with the recorded data of the sampler unit in terms of correctness by use of the indexing system and time stamp information.

Section 1 of Fig. 1 shows a 35 W solar panel, connected to a solar charge controller. The solar panel and solar controller will charge a 12 volt 7.2 Ah battery during the day. The battery will be drained during the night with the help of a static load. To create measurable results for the sampler unit, a voltage and current transducer will be connected to the solar panel. The surface temperature of the solar panel will also be measured.

The Sampler Unit in Section 2 of Fig. 1 consists of an Arduino Mega 2560 microcontroller equipped with a GPS and SD card for data storage. Values stored on the SD card will later be used as a reference to the stored values of Section 5. The sampler unit will also forward the sampled data to the Arduino Mega 2560 microcontroller in Section 3 via industry standard Modbus serial data communication protocol.

The WiFi Transmitter Unit (Fig. 1 Section 3A) uses an Arduino Mega 2560 microcontroller with an external ESP-01 WiFi module. At start up, the Arduino microcontroller connects and authenticates itself to an internet connected WiFi router. When data needs to be sent for cloud storage, the Arduino microcontroller will create a connection to ThingSpeak via the WiFi module using standard Hayes AT commands. Once connected to the cloud server, it can transfer the data and close the connection.

The GPRS CLASS 10 Transmitter Unit (Fig. 1 Section 3B) uses an Arduino Mega 2560 microcontroller with a GPRS CLASS 10 module connected serially to comport 2. The GPRS CLASS 10 module will connect itself to the network of choice. When data needs to be sent for cloud storage, the Arduino microcontroller will establish an internet connection via the GPRS CLASS 10 module using standard Hayes AT commands. Once the internet connection is established, it will validate itself to the cloud server and send the sampled data to the cloud storage website. On completion of the data transfer it will close the connection to the cloud service.

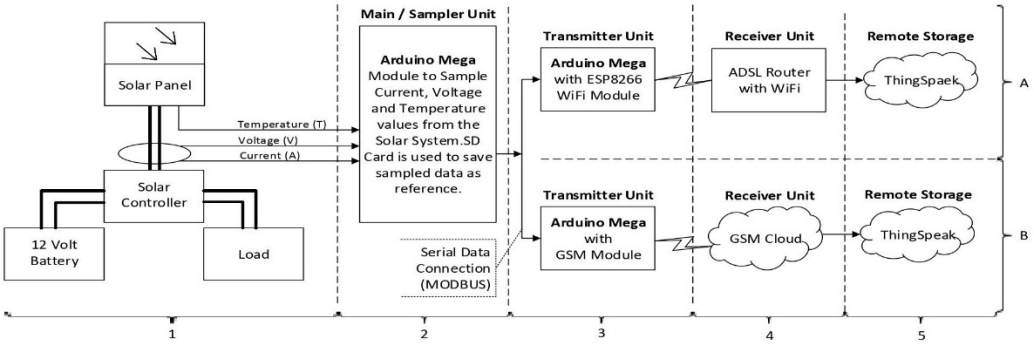


Fig. 1. Schematic diagram of evaluation of wireless technology.

The Sampler Unit samples the voltage, current and temperature of the solar system every 5 minutes for 10 days. With each sample cycle, an index will be added to the sampled data as a reference. Once the 10 days are complete, all the stored data will be collected and compared against the reference data from the SD card in the sampler unit.

The cost of all the technologies are relatively low, with LTE being the most expensive of the two, taking into account the cost of the LTE Shield for the Arduino and the cost of mobile data.

IV. RESULTS AND DISCUSSION

The outcome of the recorded data for was taken over a 10 day period from 21 September 2018 to 1 October 2018 with a total of 2900 samples.

Average, maximum and minimum time calculations were done by taking the time difference from the time received from the GPS shield on the master unit to the time stamp at the ThingSpeak cloud server when a data packet is received.

Table I shows that GPRS CLASS 10 is on average 3 seconds slower than WiFi for the transmission of data to cloud storage. It could also be perceived that the maximum time it took for GPRS CLASS 10 to get the packet of data through to the cloud server was 4 minutes and 42 seconds, to the maximum time of 21 seconds for WiFi. The minimum time for GPRS CLASS 10 was 5 seconds to the 2 second for WiFi.

TABLE I. AVERAGE, MAX AND MIN TRANSMIT TIMES OF WIFI AND GSM IN SECONDS

	WiFi	GSM
Average	4 sec	7 sec
Maximum	21 sec	282 sec
Minimum	2 sec	5 sec

Fig 2 shows the propagation delay for each sample of data for both GPRS CLASS 10 and WiFi. The response-times using different internet connections to the cloud server gave a faster response time to the WiFi connection to that of the GPRS CLASS 10 connection.

From Table I and Fig. 2 it can be seen that the transfer of data via WiFi is overall quicker than GPRS CLASS 10. This huge difference in time can mainly be attributed to the manner in which GPRS CLASS 10 and WiFi connections are treated when making a connection to the internet. When you make a TCP connection, three different parties can close the connection - the server, the client and the network. Typically, the server and client will close a connection if it has been inactive for a period of time. The network will also close the GPRS CLASS 10 connection if it has been inactive[17].

There are also various Hayes AT commands to send to the modem when data needs to be sent for cloud storage. The following is a typical data transmission cycle for GPRS CLASS 10:

- Attach to GPRS CLASS 10 service
- Bring up wireless connection
- Query current connection status
- Query IP address of the given domain name
- Start TCP connection
- Issue Send command
- Send data to cloud server
- Close the GPRS CLASS 10 connection
- Close the connection from the cloud server

All of these steps take time and requires a response from the GSM network[18]. The user needs to set timeouts and retry strategies in the software to try and accommodate all the different scenarios that may occur in the transmission of data. All of these timeouts and retry scenarios adds up and affects the successful transmission of a data packet. With these variables in mind, it must be taken into account that the WiFi module is permanently connected to the internet and requires much less steps to send data for cloud storage. A typical WiFi transmission looks as follow:

- Query IP address of the given domain name
- Start TCP connection
- Issue Send command
- Send data to cloud server
- Close the connection
- Close the connection from the cloud server

Once again the user needs to adhere to different scenarios, retry strategies and timeouts.

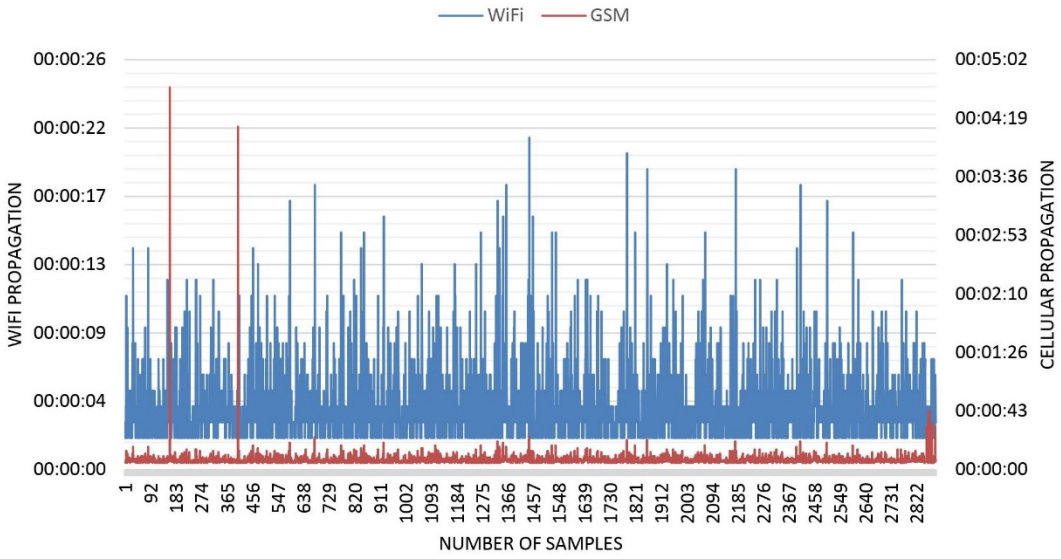


Fig. 2. Data Transmit Delay Times to ThingSpeak (2900 Samples taken over 10 days every 5 minutes).

Fig 3 shows the total amount of data packet lost over the 10 day period. It can be calculated that WiFi has a 99.96% reliability compared to 99.82% reliability of GPRS CLASS 10, thus giving a worst case packet loss of 0.18%. From this it can be accepted that packet loss does occur over both GPRS CLASS 10 and WiFi links, but the incidents is relatively rare and hard to quantify. Other studies showed that the expected packet loss over GPRS CLASS 10 is about 0.23%[19].

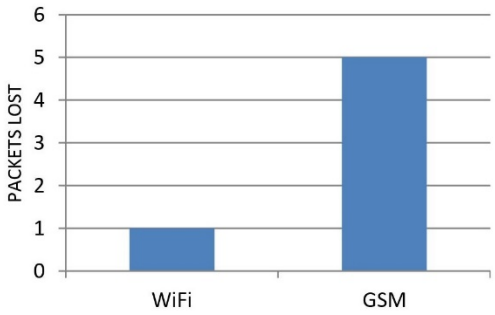


Fig. 3. Total Packets Lost.

V. CONCLUSIONS

The aim of this paper was to investigate the propagation delay and data integrity of transmitted data from cellular and WiFi networks to cloud storage. Results indicate that both WiFi and GPRS are very reliable with a maximum packet loss of 0.18% for GPRS CLASS 10 and only 0.04% for WiFi.

The transmission times to send data for cloud storage was also much higher for GPRS CLASS 10 with an average time of 7 seconds compared to that of 4 seconds for WiFi. It must be taken into account that if small amounts of data needs to be sent, that WiFi might be a more expensive option to that of a cellular device although the cellular hardware might be slightly more expensive to that of WiFi.

WiFi does have its limitations in terms of coverage footprint, the amount on active connections and security where cellular is a much more secure connection to the internet. The downside of cellular, on the other hand, is that each device requires its own cellular internet connection and if there is no cellular coverage the user cannot create the necessary infrastructure.

From the quote of Marc Benioff it is true that a combination of wireless technologies unlocks an incredible opportunity to connect everything together in a new way that is dramatically transforming the way we live and work. The results of this study suggests that researchers, hobbyists and commercial users can reliably make use of WiFi and cellular technologies to remotely transmit their data to a cloud server, as propagation delays are negligible and data integrity is ensured.

REFERENCES

[1] M. Benioff, "The world is being re-shaped by the convergence of social, mobile, cloud, big data, community and other powerful forces. The combination of these technologies unlocks an incredible opportunity to connect everything together in a new way and is drama... -." [Online]. Available: https://www.brainyquote.com/quotes/marc_benioff_532180. [Accessed: 11-Oct-2018].

[2] H. S. Dhillon, H. Huang, and H. Viswanathan, "Wide Area Wireless Communication Challenges for the Internet of Things," *IEEE Commun. Mag.*, vol. 55, no. 2, pp. 168–

174, 2017.

[3] C. P. Kruger, G. P. Hancke, S. Networks, and H. Kong, "Rapid Prototyping of a Wireless Sensor Network Gateway for the Internet of Things Using o ff -the-shelf Components," *2015 IEEE Int. Conf. Ind. Technol.*, pp. 1926–1931, 2015.

[4] M. Rouse and I. Wigmore, "What is Internet of Things (IoT)? - Definition from WhatIs.com," *IoT Agenda*, 2016. [Online].Available: http://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT. [Accessed: 11-Oct-2017].

[5] C. Wang, S. S. M. Chow, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for secure cloud storage," *IEEE Trans. Comput.*, vol. 62, no. 2, pp. 362–375, Feb. 2013.

[6] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Internet of Things," *Int. J. Commun. Syst. Int. J. Commun. Syst*, vol. 25, no. 25, 2012.

[7] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Futur. Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.

[8] Z. K. Zhang, M. C. Y. Cho, C. W. Wang, C. W. Hsu, C. K. Chen, and S. Shieh, "IoT security: Ongoing challenges and research opportunities," in *Proceedings - IEEE 7th International Conference on Service-Oriented Computing and Applications, SOCA 2014*, 2014, pp. 230–234.

[9] A. M. Gibb, "New Media Art, Design, and the Arduino Microcontroller: a Malleable Tool," *History*, no. February, p. 70, 2010.

[10] P. Thapliyal, M. S.-I. Journal, and U. 2015, "A image encryption scheme using chaotic maps," *pdfs.semanticscholar.org*.

[11] Wikipedia, "List of Arduino Compatible Boards," *Wikipedia*, 2015. [Online]. Available: https://en.wikipedia.org/wiki/List_of_Arduino_boards_and_compatible_systems#Arduino-compatible_boards. [Accessed: 15-Oct-2018].

[12] J. Chan and S. Pannerselvam, "Learn 5 Single Board Computer: Raspberry Pi, Asus Tinkerer Board, Banana PI M2, Pine A 64, Chip, Rock 64," 2018.

[13] M. Kowsigan, K. Induja, … S. K.-… and C. (I2C2), and undefined 2017, "An optimal automatic cooling system in cloud data center," *ieeexplore.ieee.org*.

[14] S. Zafar, G. Miraj, R. Baloch, … D. M.-, T. & Applied, and undefined 2018, "An IoT Based Real-Time Environmental Monitoring System Using Arduino and Cloud Service," *etasr.com*.

[15] N. Ya'acob, M. Zolkapli, … J. J.-E., and undefined 2017, "UAV environment monitoring system," *ieeexplore.ieee.org*.

[16] N. Latinovic and A. Pešic, "Architecting an IoT-enabled platform for precision agriculture and ecological monitoring : A case study ˇ arko Zec ˇ o Krstajic," *Comput. Electron. Agric.*, vol. 140, pp. 255–265, 2017.

[17] D. VYAS and D. PANDYA, "EVALUATING GPRS TECHNOLOGY FOR M2M APPLICATIONS," *researchgate.net*.

[18] V. Babu, D. S.- IJSEAT, and U. 2015, "Embedded wireless data transfer to cloud for agriculture application based on GPRS," *ijseat.com*.

[19] M. Cao and J. Fang, "Design of Remote Terminal of Air Compressor Based on STM32 and GPRS," *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 394, p. 032113, 2018.

Annexure G: Paper Presented at SATNAC 2019 Conference

Reliability and Transmission Delays of WiFi, GPRS and Radio Networks from IoT devices to cloud storage

William Benjamin van der Merwe[1], Pierre Hertzog[2] and James Swart[3]

Department of Electrical, Electronic and Computer Engineering, Central University of Technology, Free State

Private Bag X20539, Bloemfontein, 9300, South Africa

[1]vdmerwwb@eskom.ac.za,

[2]phertzog@cut.ac.za

[3]aswart@cut.ac.za

Abstract—**The internet and cloud storage is becoming more and more important to researchers, hobbyists and commercial developers. This includes the transmission of reliable data as the availability and functionality of remote sensors and IoT devices are becoming more common. The availability of high-speed internet connections, like fibre optic cable, LTE and digital radios, changed the playing field and enabled the user to transmit data to cloud storage in a very fast manner. With these various technologies available, the question now arises which technology is more reliable and efficient for IoT sensors and for users to transmit data to a cloud server. This paper aims to investigate the reliability and transmission delay of transmitted data from WiFi, GPRS and digital radio networks to cloud storage. A sampling unit was designed to evaluate analogue inputs periodically and send the recorded data to the three technologies under test. It also records the data to an on-board micro SD card along with an indexing system. The systems then transmit the sampled data and index to a cloud storage server via the communication technologies under test. The cloud-stored data is then compared with the recorded data of the sampler unit to determine data integrity. Transmission delays can be calculated by using the cloud storage server's time stamp information and the original time stamp of each data message. The outcome of this research may help researchers, hobbyists and commercial developers to make a better-informed decision about the technology they wish to use for their particular project.**

Keywords— **Wireless Fidelity (WiFi), Long Term Evolution communications (LTE), GPRS CLASS 10, Digital Radio, XBee, Cloud Storage**

I. Introduction

"*Share your IoT experiences. Share your successes, best practices, and failures. We know that IoT does not work perfectly every time, but we can learn from each other's mistakes and vow not to repeat them.*" - Maciej Kranz [1]. The past 20 years have seen progress from early internet-enabled PC's to modern mobile communication devices as well as individual devices that can connect to the internet, where both the successes and failures have been shared. Thus, the internet, in conjunction with the Internet of Things (IoT), has enabled one to register many devices on cloud services where specific data can be recorded [2]. The essence of these IoT devices is the ability to create a path for data to flow from various locations to a cloud server. Cloud storage provides researchers with the ability to access accumulated data from many platforms, anywhere in the world, with the added advantage that the data is always backed up [3].

Sensors and transducers used by researchers, hobbyists and commercial developers measure the results of experiments and investigations. This accumulated IoT data needs to be transmitted to a cloud storage database via the internet using some type of communication medium. Thus, it could be asked, which is the most reliable and efficient method to transmit sensor data through the internet to a cloud storage platform. The aim of this paper is to investigate the reliability and transmission delays of WiFi, GPRS and digital radio networks to an internet cloud storage database.

This paper firstly covers a brief literature review of the IoT, followed by the proposed research method. Lastly the results and conclusion will be discussed.

II. Literature Study

The progression of an innovative period of the IoT is upon us [4]. Nearly every industry will be affected by the IoT. In a very simple way to put it, you have "Things" that sense and collect data and send it to a cloud storage server via the internet. The IoT is the extension of the internet into physical sensor devices and everyday objects. It offers mankind the capability to measure, accumulate and analyse exchanged data instantaneously. The ability of sensors, transducers and intelligent devices to communicate seamlessly with a network creates the IoT [5]. For example, smartphones with installed sensors can collect data and information about the immediate environment like temperature, atmospheric pressure and humidity. A cloud storage database provides the platform for these sensors to store their information. This information can be used and shared between developers, scientists and architects to create a better environment for mankind [6].

Taking this into account, one must remember that large amounts of private information may also be vulnerable to cyber-crime, giving rise to security issues. However, connecting IoT devices directly to the internet is growing at an alarming pace. "What is missing is a well-architected system that extends the functionality of the cloud and provides seamless interplay among the heterogeneous components in the IoT" [7]. Added to this is not only the need to get the data to the cloud storage server as quickly as possible but also the need to ensure data integrity, which has been defined as the assurance that the data received is a replica of what was originally transmitted by an authorized entity [8].

III. Proposed Research Method

This research aims to determine the most reliable way for researchers, hobbyists and commercial developers to transmit data from a strategic location, via sensors, to an internet cloud server using various wireless technologies. These include WiFi, GPRS CLASS 10 and digital radio.

A. Overview

The research uses two types of technologies to save data to a cloud service for storage and a radio link for local storage. The stored data from each technology can then be compared to each other and to the original source data for validity, integrity and propagation delays.

The practical setup can be divided into various components:

- Solar System (Data Source)
- Main Sampler Unit and RF Transceiver
- Radio Transceivers with storage capabilities
- WiFi Transceiver Unit
- GPRS CLASS 10 Transceiver Unit

Figure 1 shows the various components of the practical setup and how they fit together.

B. Solar System

Section 1 of Figure 1 shows a 35 W solar panel, connected to a solar charge controller. The solar panel and solar controller chargers a 12 volt 7.2 Ah battery during the day. The battery is drained during the night with the help of a static load so that the solar system may again charge it the following day.

The solar system is used to create measurable analogue data for the Sampling Unit (Section 2) to measure and store as digital information. The locally stored data is later used to determine data integrity and propagation delays of the different technologies evaluated in this research.

C. Main / Sampler Unit

The main or sampler unit is the core of the system. The sampler unit scans the solar system for analogue information (Current, Voltage and Temperature). The unit is based on an Arduino Mega microcontroller board equipped with the ATmega 1280 microprocessor. It has 4 UARTs (hardware serial ports) and is used for communication, data transmission to the technologies under test as well as debugging. It also has 16 analogue inputs with a 10-bit resolution to monitor the voltage and amperage of the solar system. There are also 54 digital inputs/outputs available to monitor or control external devises. It runs from a 16 MHz crystal oscillator with a USB connection for programming and monitoring. It has a power connector (6-20VDC) and an ICSP header. The ICSP header is used as a communication path to the on-board micro SD card storage device for data logging.

Consecutively, every 5 minutes, the WiFi, GPRS CLASS 10 and digital radio receiver units are updated with new data to send to their respective storage locations. The microcontroller generates a unique index for every event sequence of analogue data that needs to be stored. This index value is used to keep track of the data throughout the system. For each new event, the date, time, index and analogue data is stored to the on-board micro SD card of the main unit. It also updates all the evaluation technologies with the most recent information using industry standard MODBUS serial communications protocol.

The WiFi and LTE sections (Figure 1 row B and C) receive their data to transmit via a hardwired serial cable from the Sampler Unit. The Sampler Unit sends a specific data bit in the message to the remote units, letting them know that they must report the information in that data stream to the cloud service for storage.

The data connection in Figure 1 row A, shows a radio connection between the sampling unit and a storage unit, using XBee 2 mW RF transceivers. The radio connection uses the same method as the WiFi and LTE to identify when a data stream needs to be stored by setting a specific bit in the data message.

XBee modules allows for very reliable and simple communication between microcontrollers, computers or anything with a serial port. Point to point and multi-point networks are supported and easy to setup. The Sampler Unit's XBee transceiver is mounted on top of the microcontroller and gets its data from the MODBUS port. It then transmits the MODBUS data to the storage microcontroller. The storage

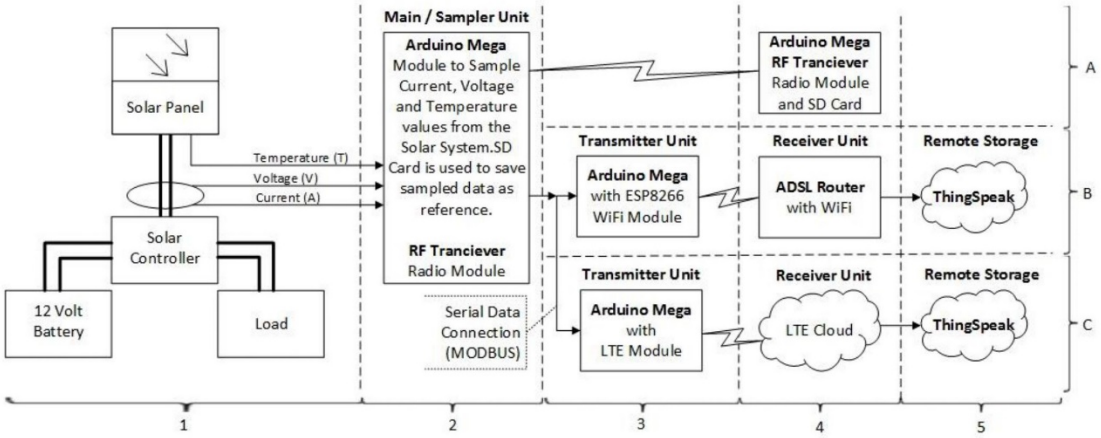


FIGURE 1: SCHEMATIC DIAGRAM OF EVALUATION OF WIRELESS TECHNOLOGY.

microcontroller in Figure 1 Section 4A stores the received data to SD card and replies back to the Sampler Unit via its own XBee transceiver.

1. Analogue Sampling

The voltage of the solar panel is measured by a small voltage module that uses a potential divider to reduce the input voltage by a factor of 5. This allows one to use the analogue input of the microcontroller to monitor voltages much higher than it is capable of sensing. For a 0 V – 5 V analogue input range one can measure a voltage of up to 25 V. The typical Open Circuit Voltage (Voc) output of the solar panel exposed to full sun is 21.6 V.

The current of the solar panel is measured by a small current sensing module that uses the ACS714 Hall effect-based linear current sensor that is mounted on a board with supporting components. It can accept a bidirectional current input with a magnitude of ± 5 A and outputs a potential voltage of 185 mV/A centred at 2.5 V with a typical error of ± 1.5 %. The typical Short Circuit Current (Isc) output for the solar panel fully exposed to the sun is 2.21 A.

The temperature sensor is fitted to the solar panel with epoxy for maximum efficiency. It uses the DS18B20 temperature sensor with an accuracy of ± 0.5 °C from -10 °C to +85 °C. It provides a 9 to 12-bit selectable temperature resolution with a unique identifiable 64-bit address ID over a 1-Wire interface.

2. GPS

The GPS is mainly used for timing and to time stamp every event cycle. The shield is compatible with the Arduino and Arduino MEGA boards. It is based on the Ublox NEO-6M receiver module with an additional SD card interface. The GPS is serially connected to the Arduino Mega's comport one and updates its data every second. The GPS fix data, as well as the analogue input data, is stored on the on-board micro SD card with every cyclic system update.

3. MODBUS

MODBUS was chosen as a communication protocol to all the remote units, as it is an industry standard protocol used by most PLC's and SCADA equipment. The protocol was created by Modicon (now Schneider Electric) in 1979. It remains the most widely available protocol for connecting industrial devices. The MODBUS protocol specification is openly published and the use of the protocol is royalty-free.

MODBUS protocol is defined as a master/slave protocol, meaning a device operating as a master (Main or Sampler Unit) will poll one or more devices operating as a slave (WiFi, GPRS CLASS 10 and Xbee Units). This means that slave devices cannot offer information to the master; it must wait until the master asks for it. The master device will write data to a slave device's register's and read data from the slave device's registers.

The most commonly used form of MODBUS protocol is MODBUS RTU. MODBUS RTU is a relatively simple serial protocol that can be transmitted via traditional UART technologies, like the Arduino Mega microcontroller. Data is transmitted in 8-bit bytes, one at a time at 19200 baud.

A typical MODBUS RTU network has one master and one or multiple slave devices. Each slave has a unique 8-bit address. Data packets sent by the master device include addresses of the slave that the message is intended for. The slave will only respond when it recognizes its own address. Responses from the slave must be in a certain time period or the master will detect it as a "no response" error.

Each data exchange request from the master is followed by a response from the slave. Each data packet, whether it is a request or response, begins with the slave address, followed by a function code and parameters defining what is being asked for or provided to the slave. A checksum at the end of the message will verify the integrity of the data packet. If the checksum does not calculate to the correct result, the slave will discard the message and the master will identify the data packet as a "no response". Normally the master will then try to resend the data packet for a predefined number of times. The general outline of each request and response is illustrated in Table I.

TABLE I. GENERAL OUTLINE FOR REQUEST AND RESPONSE MESSAGE

Address	Func Code	Reg Num	Reg Count	Data	CRC

MODBUS data is typically read and written as "registers" which are 16-bit pieces of data. The most common register is known as Holding Registers and can be read or written too. Another type of register is an Input Registers, which is read-only. The function code determines the type of register addressed by a MODBUS data packet. The most common function codes include type 3 "read holding registers". Function code 6 is used to write a single holding register and function code 16 is used to write to one or more holding registers. Only function code 16 (Preset Multiple Registers) was used in this study. The raw data in Table. II illustrates a typical MODBUS request message from the master device to a slave device to pre-set multiple registers. Both hexadecimal and decimal values are shown with the checksum at the bottom of the table.

TABLE II. MODBUS REQUEST MESSAGE DECONSTRUCTED

Description	**Hex Value**	**Decimal Value**
1st Byte - Device Address	0x01	01
2nd Byte - Function Code	0x10	16
3rd Byte – Address of the First Register – Hi Byte	0x00	00
4th Byte – Address of the Second Register – Lo Byte	0x00	00
5th Byte – Number of Registers – Hi Byte	0x00	00
6th Byte – Number of Registers – Lo Byte	0x0C	12
7th Byte – Number of Bytes to follow	0x18	24
8th Byte – Value – Hi Byte	0x00	00
9th Byte – Value – Lo Byte	0x00	00
nth Byte – Value – Hi Byte	0x00	00
nth Byte – Value – Lo Byte	0x00	00
Checksum - CRC	0xA9	169
Checksum - CRC	0xAD	173

Transmit message from master to slave in hexl: 0x01 0x10 0x00 0x00 0x00 0x0C 0x18 0x00 0x00 0xE9 0x40 0x07 0xE3 0x00 0x04 0x00 0x1B 0x00 0x0B 0x00 0x3B 0x00 0x0C 0x07 0xBD 0xFF 0xD7 0x15 0xB4 0x00 0x00 0xA9 0xAD

If the 1st Register (8th & 9th Byte) is set to a value of 1, it indicates to the slave device that the register data that follows must be used as valid data. It is then stored on-board or sent to the cloud for storage.

The raw data in Table III illustrates a typical MODBUS response message from the slave device to the master device to pre-set multiple registers. The device address always occurs first followed by the function code.

TABLE III. MODBUS RESPONSE MESSAGE DECONSTRUCTED

Description	Hex Value	Decimal Value
1st Byte - Device Address	0x01	01
2nd Byte - Function Code	0x10	16
3rd Byte – Address of the First Register – Hi Byte	0x00	00
4th Byte – Address of the Second Register – Lo Byte	0x00	00
5th Byte – Number of Registers – Hi Byte	0x00	00
6th Byte – Number of Registers – Lo Byte	0x0C	12
Checksum - CRC	0xC0	192
Checksum - CRC	0x0C	12

Transmit message from slave to master in hexl: 0x01 0x10 0x00 0x00 0x00 0x0C 0xC0 0x0C

Table IV is a description of all the data registers that the master device can set on the slave devices, including the time and temperature data.

TABLE IV. SETTABLE SLAVE DATA REGISTER

Register	Description
A0	Indicates to Slave device that the following data needs to be saved
A1	Unique index to for each data message that needs to be saved
A2	Year value from the GPS Receiver
A3	Month value from the GPS Receiver
A4	Day value from the GPS Receiver
A5	Hour value from the GPS Receiver
A6	Minute value from the GPS Receiver
A7	Second value from the GPS Receiver
A8	Voltage value of the Solar System
A9	Current value of the Solar System
A10	Temperature of the Solar panel
A11	Remote soft restart for slave devices

D. XBee

The XBee Series 2 modules allow for a very simple and reliable communication between microcontrollers, computers, systems or really anything that uses a serial port to communicate. It supports point to point and point to multi-point networks. They allow one to create very complex mesh networks based on the ZigBee mesh firmware.

Instead of being hard wiring between the master unit and the slave unit, the wiring is replaced with an XBee radio transceiver. An XBee unit is serially connected to the master unit's communication port that sends MODBUS polls to the slave units.

At the slave unit, an XBee radio transceiver is serially connected to the microcontroller's serial port. The microcontroller decodes the serially received MODBUS data and responds to the master with an acknowledgement message. Each time that the slave unit detects that the first register is set, it will know that it needs to save the following data to the on-board micro SD card for data verification.

E. WiFi

The WiFi microchip used in this study is the ESP8266 microcontroller and supports the full TCP/IP stack. The chip first came to light in August 2014 with the ESP-01 produced by a third-party manufacturer, Ai-Thinker.

The WiFi-microcontroller is hard-wired to the main controller via one of its serial communications ports. The master controller continuously sends MODBUS Preset Register Requests to the WiFi-microcontroller updating it with new information. As soon as the WiFi-microcontroller detects that the first register is set to a value of 1, it immediately acknowledges the master unit and starts the process of sending the remaining information of the data packet to the cloud for storage.

The WiFi module is permanently connected to the internet via a WiFi Router. The microcontrollers connect serially with its second serial communications port to the WiFi module using Hayes AT commands.

AT commands are then used to send the validated data via the internet to ThingSpeak (discussed on the next page). A typical WiFi transmission session involves the following:

- Query IP address of the given domain name
- Start TCP connection
- Issue Send command
- Send data to a cloud server
- Close the connection
- Close the connection from the cloud server

The software of the WiFi-microcontroller needs to set timeouts and retry strategies in the software to try and accommodate all the different scenarios that may occur in the transmission of data. All of these timeouts and retry scenarios adds up and affects the total transmission time of a successful transmission of a data packet.

F. GPRS CLASS 10

The SIM900 is a complete Quad-band GSM solution. With its industry-standard interface, the SIM900 works on a frequency of 850/900/1800/1900MHz and which can be used not only to access the internet but also for communication with low power consumption. It can communicate with microcontrollers via GSM 07.07, 07.05 and SIMCOM enhanced AT Commands.

The GPRS CLASS 10 microcontroller is identically connected as the WiFi-microcontroller and works on exactly the same Hayes AT Command principles. It is also continuously updated with new information from the master device using MODBUS protocol. As soon as it detects that the first data register is set to a value of 1, it will start its transfer process to send the latest data information via the internet to ThingSpeak.

The only difference between the WiFi unit and the GPRS CLASS 10 unit is that the GPRS CLASS 10 unit is not permanently connected to the internet, although it is always connected to the mobile network. For the GPRS CLASS 10 unit to send data, it must first establish a connection to the internet via the mobile network. Only after the internet connection is established can the unit start to transfer data.

Due to the GPRS CLASS 10 unit not been permanently connected to the internet, it must use a couple of extra steps in the GPRS CLASS 10-microcontrollers software to make a connection to the internet. This must be done before data can be sent for cloud storage.

A typical GPRS CLASS 10 transmission session involves the following:

- Attach to GPRS CLASS 10 service
- Bring up a wireless connection
- Query current connection status
- Query IP address of the given domain name
- Start TCP connection
- Issue Send command
- Send data to a cloud server
- Close the GPRS CLASS 10 connection
- Close the connection from the cloud server

The software of the GPRS CLASS 10-microcontroller needs to set timeouts and retry strategies in the software to try and accommodate all the different scenarios that may occur in the transmission of data. All of these timeouts and retry scenarios adds up and affects the total transmission time of a successful transmission of a data packet.

G. ThingSpeak

The developers of ThingSpeak created this platform as an open source IoT application to store and retrieve data for things using it. ThingSpeak provides the facility for storage of data obtained from sensors and it also provides one with the instant visualization of the data [9]. It is very easy for the user to configure sensor devices to send their data to ThingSpeak using IoT protocols. The stored data can be viewed live from a standard web browser or on a mobile device supporting Android or IoS. Data can also be downloaded in a CSV or JSON format from a domain for offline analyses.

The web site was originally launched in 2010 as a service for IoT applications. The core element of ThingSpeak is the 'ThingSpeak Channel'. A channel stores the data that is sent to ThingSpeak with the following basic elements:

- 8 fields for storing data of any type - These can be used to store the data from a sensor or from an embedded device.
- 3 location fields - Can be used to store the latitude, longitude and elevation. These are very useful for tracking a moving device.
- 1 status field - A short message to describe the data stored in the channel.

IV. Results and Discussion

Recorded data was sampled every 5 minutes for a 209-day period from 1 October 2018 to 27 April 2019 that equals a total sample size of 59725.

Load shedding has a huge impact on our daily lives due to the interruption of the supply of electrical energy. It is enforced when there is an imbalance between the supply and demand for electrical energy. It is enforced to protect the national grid of South Africa from a complete shutdown. Load shedding has a negative impact on many electrical and electronic devices, including the operation of WiFi hotspots. The interruptions of the WiFi transmissions in this study (WiFi router dependent on a stable electricity supply) and the corresponding records of the other two technologies have been removed to present an equal result across the board.

Figure 2 shows the availability of the WiFi router and the power failure interruptions due to load shedding for the period of 1 to 6 December 2018. It is perceived from Figure 2 that there was normal stage 2 load shedding for the first 3 days where the electrical energy supply was interrupted once a day. Then on day four, stage 4 load shedding was implemented and supply interruptions were more frequent.

FIGURE 2: WIFI ROUTER SUPPLY LOSS DUE TO LOAD SHEDDING

Propagation time delays were determined by calculating the average, maximum and minimum time deviations. These delays were calculated between the time stamp on the micro SD card (received from the GPS shield on the master unit) and the time stamp at the ThingSpeak cloud server when a data packet is received.

Table VI shows that GPRS CLASS 10 is on average 4 seconds slower than WiFi for the transmission of data to cloud storage. The radio connection was less than 0.1 second. It could also be perceived that the maximum time it took for the GPRS CLASS 10 to transmit a packet of data through to the cloud server was 4 minutes and 14 seconds (254 sec), to the maximum time of 4 minutes and 27 seconds (267 sec) for WiFi. The minimum time for the GPRS CLASS 10 was 5 seconds as compared to the 2 second for WiFi. The benefit that the XBee radio communication has over the GPRS CLASS 10 and WiFi communication is that the XBee radio does not have any internet communication overheads to adhere to, but can send its data without any delay. The drawback, however, is that the XBee radio needs a third-party communications device to connect to the internet.

TABLE VI. AVERAGE, MAX AND MIN TRANSMIT TIMES OF RADIO, WIFI AND GPRS CLASS 10 IN SECONDS

	XBee Radio	WiFi	GPRS CLASS 10
Average	>0.1sec	4 sec	8 sec
Maximum	>0.1 sec	267 sec	254 sec
Minimum	>0.1 sec	2 sec	5 sec

Figure 3 shows the propagation delay for each sample of data for both GPRS CLASS 10 and WiFi. The response-times using different internet connections to the cloud server gave a faster response time to the WiFi connection to that of the GPRS CLASS 10 connection. This is mainly due to the manner in which GPRS CLASS 10 and WiFi connections are treated when making a connection to the internet. The number of steps to make a GPRS CLASS 10 connection, send the data and close the connection is about a third more than with WiFi. Each of the communication steps taken to transmit a data packet to cloud storage via the internet takes time and a timeout period needs to follow, to assure successful data transmission for both WiFi and GPRS CLASS 10.

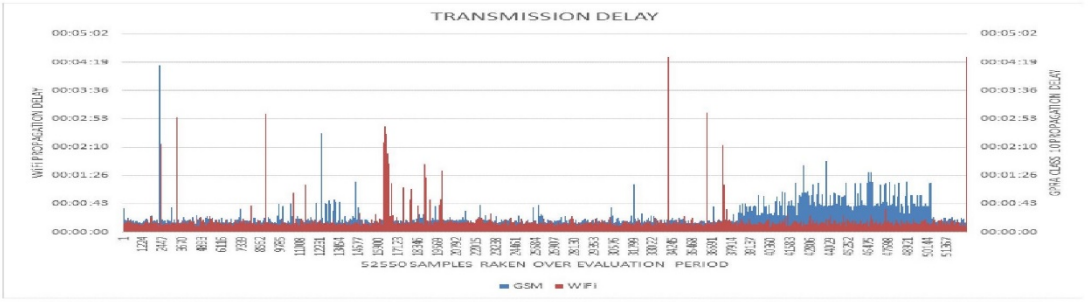


FIGURE 3: DATA TRANSMIT DELAY TIMES TO THINGSPEAK (52550 SAMPLES TAKEN OVER 209 DAYS EVERY 5 MINUTES).

Figure 4 shows the total amount of data packets lost over the 209-day period. Calculating the reliability based on this figure indicates that the XBee radio system is much more reliable with 0.21% data packets lost, closely followed by WiFi with 0.31% and GPRS CLASS 10 with 1.46%. From this, it can be accepted that packet loss does occur for all three technologies under test. However, data loss incidents is relatively rare and hard to quantify. Other studies showed that the expected packet loss over GPRS CLASS 10 is about 0.83% [10].

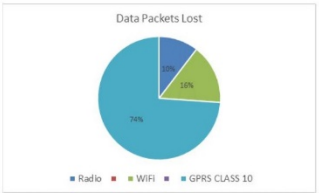


FIGURE 4: TOTAL DATA PACKETS LOST

V. Conclusions

The aim of this paper was to investigate the reliability and transmission delay of transmitted data from WiFi, GPRS and digital radio networks to cloud storage. Results indicate that all three technologies are very reliable with a maximum packet loss of 1.46% (872 out of 59727) for GPRS CLASS 10, 0.31% (185 out of 59727) for WiFi, and only 0.21% (123 out of 59727) for XBee radio.

The average transmission time for GPRS CLASS 10 was twice that of WiFi with digital radio have negligible delays. It might be considered that if a small amount of data needs to be sent for data storage, that digital radio and WiFi might be a more expensive option to GPRS CLASS 10, due to external line rentals. Digital radio might be a good option if various sensors are collecting data, and sending their information via digital radio to a data concentrator. The data concentrator can then be connecting to the internet with WiFi or GPRS CLASS 10. On the other hand, WiFi can be used instead of digital radio to connect sensors to a data concentrator, but the limitation of WiFi is normally the coverage footprint.

From the quote of Maciej Kranz at the outset, one needs to investigate the performance of IoT and communication technologies more and more. With the knowledge gained in this study, it is recommended that researchers, hobbyists and commercial developers can reliably make use of digital radio, WiFi and cellular technologies to remotely transmit their sensor data to a cloud server, as transmission delays are negligible and data integrity is ensured.

References

[1] M. Kranz, "Share your IoT experiences. Share your successes, best practices, and …" [Online]. Available:

[2] H. S. Dhillon, H. Huang, and H. Viswanathan, "Wide Area Wirelss Communication Challenges for the Internet of Things," *IEEE Commun. Mag.*, vol. 55, no. 2, pp. 168–174, 2017.

[3] C. P. Kruger, G. P. Hancke, S. Networks, and H. Kong, "Rapid Prototyping of a Wireless Sensor Network Gateway for the Internet of Things Using o ff -the-shelf Components," *2015 IEEE Int. Conf. Ind. Technol.*, pp. 1926–1931, 2015.

[4] Z. K. Zhang, M. C. Y. Cho, C. W. Wang, C. W. Hsu, C. K. Chen, and S. Shieh, "IoT security: Ongoing challenges and research opportunities," in *Proceedings - IEEE 7th International Conference on Service-Oriented Computing and Applications, SOCA 2014*, 2014, pp. 230–234.

[5] A. M. Gibb, "New Media Art, Design, and the Arduino Microcontroller: a Malleable Tool," *History*, no. February, p. 70, 2010.

[6] P. Thapliyal, M. S.-I. Journal, and U. 2015, "A image encryption scheme using chaotic maps," *pdfs.semanticscholar.org*.

[7] Wikipedia, "List of Arduino Compatible Boards," *Wikipedia*, 2015. [Online]. Available: https://en.wikipedia.org/wiki/List_of_Arduino_boards_and_compatible_systems#Arduino-compatible_boards. [Accessed: 15-Oct-2018].

[8] J. Chan and S. Pannerselvam, "Learn 5 Single Board Computer: Raspberry Pi, Asus Tinkerer Board, Banana PI M2, Pine A 64, Chip, Rock 64," 2018.

[9] N. Latinovic and A. Pešic, "Architecting an IoT-enabled platform for precision agriculture and ecological monitoring : A case study ˇ arko Zec ˇ o Krstajic," *Comput. Electron. Agric.*, vol. 140, pp. 255–265, 2017.

[10] M. Cao and J. Fang, "Design of Remote Terminal of Air Compressor Based on STM32 and GPRS," *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 394, p. 032113, 2018.

William Benjamin van der Merwe is a MEng electrical engineering student at the Central University of Technology, Free State.

Prof Pierre Hertzog received his DTech in 2004 and is an Associate Professor at the Central University of Technology.

Prof James Swart received his DTech in 2011 and is an Associate Professor at the Central University of Technology