

Creating a mesh sensor network using Raspberry Pi and XBee radio modules

By
Michael Forcella

In Partial Fulfillment of the Requirement for the Degree of

MASTER OF SCIENCE

In
The Department of Computer Science
State University of New York
New Paltz, NY 12561

May 2017

Creating a mesh sensor network using Raspberry Pi and XBee radio modules

Michael Forcella

State University of New York at New Paltz

We the thesis committee for the above candidate for the
Master of Science degree, hereby recommend
acceptance of this thesis.

David Richardson, Thesis Advisor
Department of Biology, SUNY New Paltz

Chirakkal Easwaran, Thesis Committee Member
Department of Computer Science, SUNY New Paltz

Hanh Pham, Thesis Committee Member
Department of Computer Science, SUNY New Paltz

Approved on _____

Submitted in partial fulfillment for the requirements
for the Master of Science degree in
Computer Science at the
State University of New York at New Paltz

ABSTRACT

A mesh network is a type of network topology in which one or more nodes are capable of relaying data within the network. The data is relayed by the router nodes, which send the messages via one or more 'hops' until it reaches its intended destination. Mesh networks can be applied in situations where the structure or shape of the network does not permit every node to be within range of its final destination. One such application is that of environmental sensing. When creating a large network of sensors, however, we are often limited by the cost of such sensors. This thesis presents a low-cost mesh network framework, to which any number of different sensors can be attached. The hardware configuration is detailed in such a way that anyone with a modest understanding of technology will be able to reproduce it. The software setup required by the user has also been minimized and clearly documented. Details specific to the user's setup can be entered into a configuration file and the majority of software scripts are scheduled to run automatically via Linux Cron jobs. I conclude by outlining several potential modifications to the framework, including further automation of the software setup, inclusion of additional hardware, and alternate methods for downloading data from the network.

TABLE OF CONTENTS

	Page
List of Figures	iii
List of Tables	iv
Chapter 1: Introduction	1
1.1 Environmental sensors	2
1.2 Citizen science	3
1.3 Interdisciplinary learning	4
1.4 Spatial variability in ecosystems	4
1.5 The DIY sensor.	5
Chapter 2: Assembly and configuration	7
2.1 Hardware options	7
2.1.1 Arduino and Raspberry Pi	7
2.1.2 XBee radio modules and networks	9
2.2 Setting up the Raspberry Pi	12
2.2.1 Configuring the XBee	13
2.3 Powering the Pi.	14
2.4 Attaching sensors	17
Chapter 3: Programming and setup	19
3.1 Node.js	19
3.2 MySQL	20
3.3 Reading from devices	21
3.4 Reading and writing files	22
3.5 Sending data to a remote computer	23
3.6 Cron and Bash scripting	24
3.7 XBee	25
Chapter 4: Deployment and results	26
4.1 Initial deployment setup	26
4.1.1 Nodes and roles	26
4.1.2 Data flow	27
4.1.3 Data view	28
4.2 Results and figures	29
4.3 Successes and failures	30
4.4 Further testing	31
4.4.1 Additional deployments	31
Chapter 5: Future enhancements and applications	33
5.1 As an open source framework	33
5.2 Needed improvements	34

5.2.1	Power consumption	34
5.2.2	Messenger queues	35
5.2.3	Error checking	35
5.3	Potential enhancements	36
5.3.1	Data download	36
5.3.2	Integration with public database	37
5.4	Future applications	38
5.4.1	Local issues	39
5.5	Final thoughts	40
Appendix A: README file and setup instructions		41
Appendix B: Node.js files		52
Bibliography		81

LIST OF FIGURES

Figure Number	Page
Figure 1.1: The mesh sensor network concept	6
Figure 2.1: Network topologies with ZigBee protocol	9
Figure 2.2: Raspberry Pi connected to breadboard via breakout board and ribbon cable	12
Figure 2.3: XBee wiring diagram	13
Figure 2.4: Solar array configuration	14
Figure 2.5: ADC wiring diagram	15
Figure 2.6: Temperature sensor wiring diagram	17
Figure 3.1: A simple callback function	20
Figure 3.2: Database structure	21
Figure 3.3: Cron scripting	24
Figure 4.1: Graphical display of voltage data	28
Figure 4.2: Latest readings from all available nodes	29
Figure 5.1: Hypothetical mesh network on Lake Minnewaska	39

LIST OF TABLES

Table Number	Page
Table 2.1: Comparison of Raspberry Pi and Arduino specifications	8
Table 2.2: Comparison of XBee Series 1 and Series 2 modules	10
Table 2.3: Comparison of ZigBee and DigiMesh networks	11
Table 2.4: Sensors and cost, available for Raspberry Pi	16
Table 4.1: Voltage readings from initial deployment	30
Table 4.2: Voltage drain from various deployments	32
Table 5.1: Cost, size and weight of solar components	35

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor David Richardson. Without his support I never would have gotten involved with my work in environmental sensors. He helped me to bridge the gap between my background in biology and my current studies in computer science, and for that I am very grateful. The time that I was able to spend out in the field with him was a welcome respite from far too many hours spent sitting at a desk. His work and ideas are always interesting, and a strong source of inspiration for my own future pursuits. His help with editing and organizing my thoughts was invaluable, and greatly appreciated.

I would like to thank Chirakkal Easwaran for his assistance with all things hardware and technical related. I went into this project with very little knowledge of hardware, and his expertise and interest in Raspberry Pi and Linux was a great asset. Aside from providing me with the proper knowledge and tools, he always expressed genuine interest in my work and was very helpful in keeping me motivated. I would also like to thank Andrew Pletch, without whom I fear I may have lost interest in computer science long ago. He has endless knowledge, and he was able to find the perfect balance between challenging new students and not discouraging them. It was because of his encouragement that I decided to take on the challenge of this thesis.

I would also like to thank William Dendis for his assistance with copy editing, as well as all of my other friends for their unending support and help keeping me focused and motivated.

Chapter 1

Introduction

We live in uncertain times. The list of environmental problems seems to grow, while the solutions feel increasingly out of reach. But daunting as it may be, it is imperative that we address these issues. The earth is, in the words of Carl Sagan, “the only home we've ever known.” To not focus our energies on its continued health would be folly and extremely irresponsible both to ourselves and future generations.

One of the most important issue to address today is climate change. This fact becomes increasingly harder to deny as the scientific consensus closes in [1]. Climate change has the potential to cause us great harm in the very near future. It is also extremely complex, and making predictions for the future has proved both challenging and controversial. As Garrett et al. [2] pointed out in their analysis of climate change effects on plant ecosystems, making an effective prediction "requires consideration of a wide range of factors." The earth is a dynamic system with innumerable variables to be measured. Making an accurate assessment of our environment will necessitate collecting and modeling large amounts of data from many different fields of scientific research [3].

While our technical capabilities have been increasing, there are still other limitations that we face. Scientists can't be everywhere at once constantly collecting data. Recently in the United States, the politicization of the issue has led to fears that government data might be suppressed. Technology, utilized by non-specialists, those of us not formally trained, can help fill the gap. This project aims to provide a process by which individuals and groups can make meaningful contributions to the record in the form of environmental observations, gathered and recorded by affordable and accurate instruments which can be deployed by non-specialists.

1.1 Environmental sensors

New technologies, such as robotics, can allow us access to areas that might otherwise be off limits, while automated sensors enable us to take measurements "at rates hitherto impossible to achieve." [4] We now have the capability, if we chose, to take a measurement every minute for a year. This kind of monitoring gives us a much clearer picture of the world by allowing us to capture the effects of episodic or extreme events [5]. To know how a system behaves at all times provides a full picture of the system and establishes a threshold of healthy vs. unhealthy for systems that very often have a degree of natural variability.

There are many scientific organizations employing environmental sensors, monitoring everything from arctic temperatures [6] to pollution levels [7]. In measuring the health of our planet, a very important indicator is the health of our water systems, including our oceans, lakes, and rivers. In 2005, a network of limnologists, scientists studying lakes, created the Global Lake Ecological Observatory Network, or GLEON. Members of the GLEON network employ many different types of sensors, measuring lake data all around the world. The GLEON network now includes over 600 members in 51 countries [8].

Of course, simply collecting the data is not enough. The data must be interpreted, and a single scientist looking at a single lake is very limiting. By linking scientists together under a common goal, GLEON hopes to address another issue among scientists, which is that of data sharing and accessibility. All data collected by the GLEON network is available to all other GLEON members. There are also annual meetings, where scientists from around the world gather to share and collaborate.

Here in the Hudson Valley we have access to an impressive historical environmental record, thanks to the Smiley family, owners of the Mohonk Mountain House. Starting with the original owners, Albert and Alfred Smiley, the Smiley family has kept extensive records of the flora and fauna as well as

weather data going as far back as 1896 [9]. In the 1970's, the Mohonk Preserve also began collecting lake data. The lake measurements were taken twice daily, which continues to this day. Because the resources of the researchers are limited, much of this data collection is delegated to volunteers. Ensuring that these measurements are recorded accurately every day, twice a day, can be a very challenging task.

In the hopes of reducing this burden on the researchers, Mohonk Lake was recently established as a GLEON site [10]. We are now in the process of installing an extensive system of automated sensors at the lake, measuring weather and lake conditions. Once the setup is complete for the automated sensors, we will have real time data, with measurements taken every fifteen minutes, available to both the researchers and the general public. This will improve the quality of the data collected, and also free resources for other research.

1.2 Citizen science

Another important aspect of the Mohonk Preserve is the citizen science program. This involves having trained volunteers assist in collecting data on plants, animals, and climate. Citizen science is becoming critically important in many areas of research [11]. This is especially true when studying large and dynamic systems, such as the ecosystems of our own planet. One area where citizen science has had an enormous impact is on ornithology, the study of birds. Organizations such as the Audubon Society have been able to muster thousands of volunteers to record their observations through programs such as the Great Backyard Bird Count [12]. Lake ecologists hope to be able to do the same.

Members of the GLEON community have recently created Lake Observer, a mobile app for collecting lake data. The app allows anyone with a smart phone or tablet, anywhere in the world, to submit data to a public database. The app collects data related to water quality, weather, and aquatic

vegetation. By putting this tool in the hands of the general public, lake scientists will have access to data from all across the globe. Users of the app might be anyone from academics to lake enthusiasts, or even local fishermen. There are many people that stand to benefit from the health of our planet's lakes, and the Lake Observer project hopes to give them an opportunity to join in the efforts.

1.3 Interdisciplinary learning

The role of technology is becoming increasingly important for all types of science. Ecology is no exception. Mobile apps such as Lake Observer and environmental sensors can be powerful tools for scientists, but only if they know how to use them. Depending on the complexity of the technology, it will often require assistance from people specifically trained in their use, and each step in the process might require people trained in a variety of disciplines.

In the example of deploying sensors, it is unlikely that an ecologist would have a full understanding of the software and hardware involved. A programmer would need to be brought on to help manage databases and other software and programming needs. Should anything go wrong with the sensors, it is very likely that an information technologist would need to be called on to help diagnose any sort of hardware or networking issue. When the sensors are finally deployed and the data collected, it must also be modeled and presented in such a way that the scientists are able to interpret it. It can often be a challenge for people of various backgrounds to communicate when they may be using a very different set of vocabulary. For this reason, it is helpful to encourage interdisciplinary learning and collaboration early on to help narrow this gap.

1.4 Spatial variability in ecosystems

The set up for the majority of the GLEON sites involves a single buoy deployed at a lake with a

series of precise sensors attached. However there are spatial limits to data collection at a single location in a lake and it might not give a complete picture of the entire lake ecosystem. There is a certain amount of variability throughout an ecosystem [13]. For this reason, it is beneficial to take readings from multiple points in a lake. This presents somewhat of a challenge, when attempting to take measurements from multiple points but within the same time frame. One could deploy multiple sensors, but depending on the size of the lake, this might require ten or more sensors. Due to the cost of many sensors, this would be a serious limitation to many studies. The proposed framework in the following pages aims to address this cost limitation.

1.5 DIY sensor project

Environmental sensors have been used by scientists for some time now, but it is only within the last ten years or so that the cost has made them available to the average consumer. There are now thousands of hobbyists tinkering with robots and sensors, building complicated and often very functional projects [14]. The need for a low cost sensor may have been here for a long time but it is only recently that it has become feasible.

The basic concept of the mesh sensor network is illustrated in Figure 1.1. The framework for this project was built using Raspberry Pi computers, which are very small, single-board computer that can be purchased for less \$40 each. The Pis are networked together using XBee radio modules. XBee radios have the advantage of being small, inexpensive, long range, and low power. The Pis are powered by a solar array and rechargeable battery. Various sensors can be attached to the Pi, reading data into a database, which is then passed through the network to a main coordinator node. The data can then be downloaded in any number of ways (e.g., Wi-Fi or Bluetooth). Not including the cost of sensors, which can vary greatly depending on the sensor type, each unit can be built for about \$200. Compared with

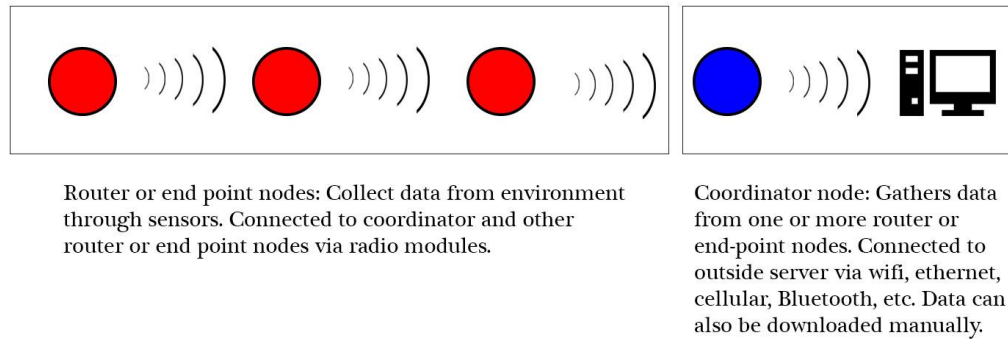


Figure 1.1: The mesh sensor network concept

the cost of many prepackaged sensors, this is an order of magnitude lower.

One of the main goals of this project is to make sensing systems available to a wider audience, which may include graduate and undergraduate researchers, or even hobbyists. Deploying ten or more sensors suddenly becomes feasible to many academics or researchers. The other objective is to encourage interdisciplinary learning. While an affordable sensor framework is possible, there are few people building such frameworks, likely because of the disconnect between ecologists and programmers.

The function of the framework would be most useful to an ecologist, but the process of building the sensors may be outside their normal comfort zone. Given time and motivation, anyone with a reasonable understanding of computers would not be able to follow the guidelines presented. However, in an ideal situation, implementation of this framework would involve biologists, who would set the goals of the project, electrical engineers to work with the hardware, and computer scientists to deal with the programming. If students and academics from different disciplines were able and willing to collaborate on a project they might find that the arrangement is both interesting and beneficial to everyone involved.

Chapter 2

Assembly and configuration

2.1 Hardware options

Much of the relevant technology that is available today was either unavailable or unaffordable ten years ago. This brings benefits and challenges. With so many options to choose from, there should be a solution to nearly any problem, and having numerous ways to address a problem gives us the freedom to find a solution that best fits our needs and resources.

However, having more technology increases the learning curve and requires constant vigilance to stay current. In some cases, particularly with new technology, the documentation might be limited, making implementation and learning difficult. Having numerous choices also requires a considerable amount of time spent weighing options. There is also the risk of investing time and money into a piece of equipment only to find that it is not well suited to its intended purpose. Similarly, with the rapidly changing technological environment, it is possible that some new technology might come along to make some piece of equipment currently being employed impractical or inferior, forcing an additional time or monetary investment into the project.

2.1.1 Arduino and Raspberry Pi

In selecting a base processor for this project there were two main choices, the Arduino and the Raspberry Pi. Both are small (about the size of a pack of a smartphone), affordable, and capable of interfacing with a wide variety of hardware, and both are frequently used for remote sensing (see Table 2.1 for comparison). The Arduino is a micro-controller, not a full computer, which means it is capable of running a single program, over and over again. The Arduino uses its own specific

Table 2.1: Comparison of Raspberry Pi and Arduino specifications

	Raspberry Pi 2	Raspberry Pi 3	Arduino Uno R3
Price	\$35.00	\$35.00	\$30.00
CPU	900 MHz	1.2 GHz	16 MHz
RAM	1 GB	1 GB	2 KB
Storage	microSD	microSD	32 KB
Operating System	Linux	Linux	n/a
I/O Pins	17 Digital	17 Digital	14 Digital, 6 Analog
USB Ports	4	4	1
On-Board Ethernet	Yes	Yes	n/a
Video Out	HDMI	HDMI	n/a
Input Voltage	5v	5v	7-12v
Minimum Power	800 mA (3.5W)	800 mA (3.5W)	42 mA (0.3W)
Size	3.37"x2.125"	3.37"x2.125"	2.95"x2.10"

programming language and has a reputation for its simplicity and ease of use. It is popular with roboticists and hobbyists, and for that reason it tends to be very well documented. Gandra et al. [15] were able to build a very affordable framework for ecological sensing using Arduino.

I elected to use the Raspberry Pi for this project because it would allow for a wider variety of programming options. Raspberry Pi differs from Arduino in that it runs a full Linux operating system, making it a fully functional computer rather than simply a device for interfacing with other hardware. Having a full operating system offers somewhat more power and freedom compared with the Arduino. The Arduino and the Raspberry Pi are able to interface with each other, so it is possible to use both in a single project, perhaps having the Arduino interface with the sensors and then having the Pi handle the data processing and computations. While the Raspberry Pi might require more initial configuration than an Arduino, it also allows for the possibility of automating a lot of user setup using bash and Cron scripts.

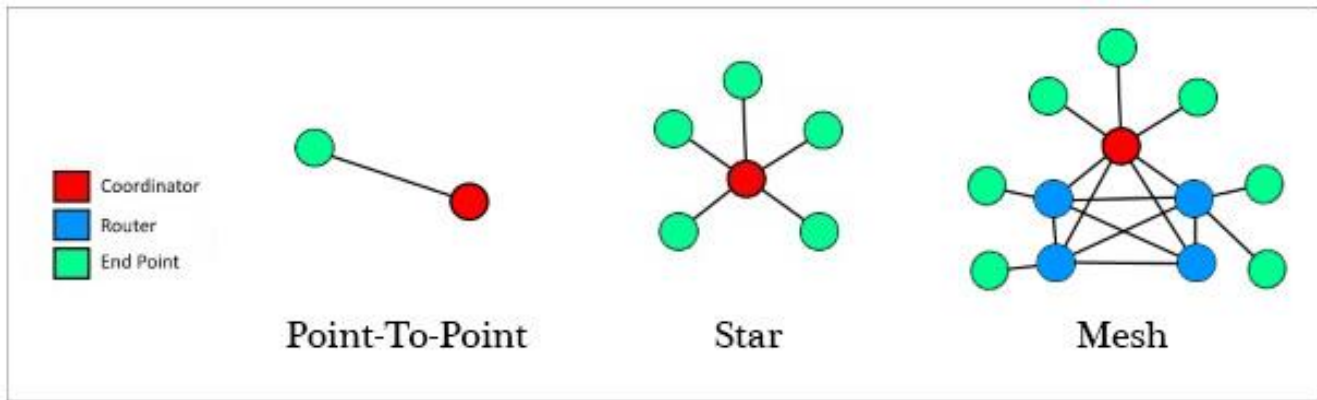


Figure 2.1: Network topologies with ZigBee protocol

2.1.2 XBee radio modules and networks

XBee is a series of radio modules. These modules transmit serial data and are able to interface with the Raspberry Pi's GPIO port. There are also USB adapters available which allow the XBee to plug directly into the Pi's USB port. There were two different models considered, the Series 1 and Series 2. They each have differences in communication protocols, network topology, range, and power consumption (see Table 2.2).

The first two things that one should consider are range and power consumption. In a side-by-side comparison, the S1 and S2 both perform comparably, and reasonably well, for our particular need. Power consumption is minimal, and range is up to 120 meters in ideal conditions. It should be noted that the Series 1 and Series 2 are both available in "Pro" versions, which have a much extended range, up to one mile. However, they also consume significantly more power, about 60mW compared with just 2mW for the regular modules. Since we were trying to limit power consumption, and because the normal range was enough for most situations, the Pro versions were not considered for use.

The second point considered, and the main difference between the S1 and S2 modules, was the type of network protocol that they employed. The different types of topologies available with the XBee

Table 2.2: Comparison of XBee Series 1 and Series 2 modules

	XBee Series 1	XBee Series 2
Network Protocol	802.15.4 or DigiMesh	ZigBee Mesh
Indoor/Urban range	up to 100 ft. (30m)	up to 133 ft. (40m)
Outdoor, line-of-sight range	up to 300 ft. (100m)	up to 400 ft. (120m)
Transmit Power Output	1 mW (0dbm)	2 mW (+3dbm)
Receiver Sensitivity	-92dBm (1% PER)	-98dBm (1% PER)
Network Topologies	Point to point, Star, Mesh (with DigiMesh firmware)	Point to point, Star, Mesh

are: point-to-point, star, and mesh (see Figure 2.1). In a point-to-point network, each module is configured to communicate with another single module based on its specific address. A module can be reinitialized to communicate with a different device, but the modules are paired with one, and only one, other device. A module can also be programmed to broadcast to every other module in range. This is the star topology, with several modules all connected to one central node (Figure 2.1). The third type of topology, and the one used here, is the mesh network (Figure 2.1).

A mesh network is one where several nodes, known as routers, are able to receive data and also send that data to one or more other routers. By default, the S1 modules use the 802.15.4 protocol. This protocol only allows for point-to-point communication under normal setup or star networks via broadcast. However, there is an additional firmware available, known as DigiMesh, which does allow for mesh networking. The S2 modules, by contrast, can only run the ZigBee mesh firmware.

While both the S1 and S2 are capable of mesh networking, there are several differences in their structure and functionality (see Table 2.3). On a ZigBee network, each module must be loaded with firmware which will assign it one of three possible roles: coordinator, router, or end point. Every network must have one, and only one, coordinator, which is able to send or receive data from other

Table 2.3: Comparison of ZigBee and DigiMesh networks

	ZigBee Mesh	DigiMesh
Node Types, Benefits	Coordinators, Routers, End Devices. End Devices potentially less expensive because of reduced functionality.	One type, homogenous. More flexibility to expand the network. Simplifies network setup. Increases reliability in environments where routers may come and go due to interference or damage.
Sleeping Routers, Battery Life	Only End Devices can sleep.	All nodes can sleep. No single point of failure associated with relying on gateway or coordinator to maintain time synchronization.
Frame Payload, Throughput	Up to 80 bytes.	Up to 256 bytes, depending on product. Improves throughput for applications that send larger blocks of data.
Supported Frequencies and RF Data Rates	Predominantly 2.4 GHz (250 Kbps). 900 MHz (40 Kbps) and 868 MHz (20 Kbps) not widely available.	900 MHz (10, 125, 150 Kbps). 2.4 GHz (250 Kbps).
Interoperability	Potential for interoperability between vendors, anything using the ZigBee protocol.	Proprietary, only communicates with other XBee Series 1 modules.

nodes and is responsible for maintaining the network. If the coordinator goes offline, the network will fail. Routers are able to send or receive data from other nodes. End points are only able to send data. Because of the reduced functionality of the end points, they also require less power.

On an S1 mesh network, there are no roles assigned to the nodes. Every node is equal. This gives the network some additional flexibility to expand and reconfigure if nodes are lost due to damage, interference, or power loss. There is also an advantage in power consumption as every node can enter sleep mode, whereas on an S2 network, only the end point nodes are capable of sleeping. In the end however, I chose to use the S2 modules, mainly because there were more software libraries available and the documentation was much better compared to that of the S1, specifically in terms of programming a mesh network.

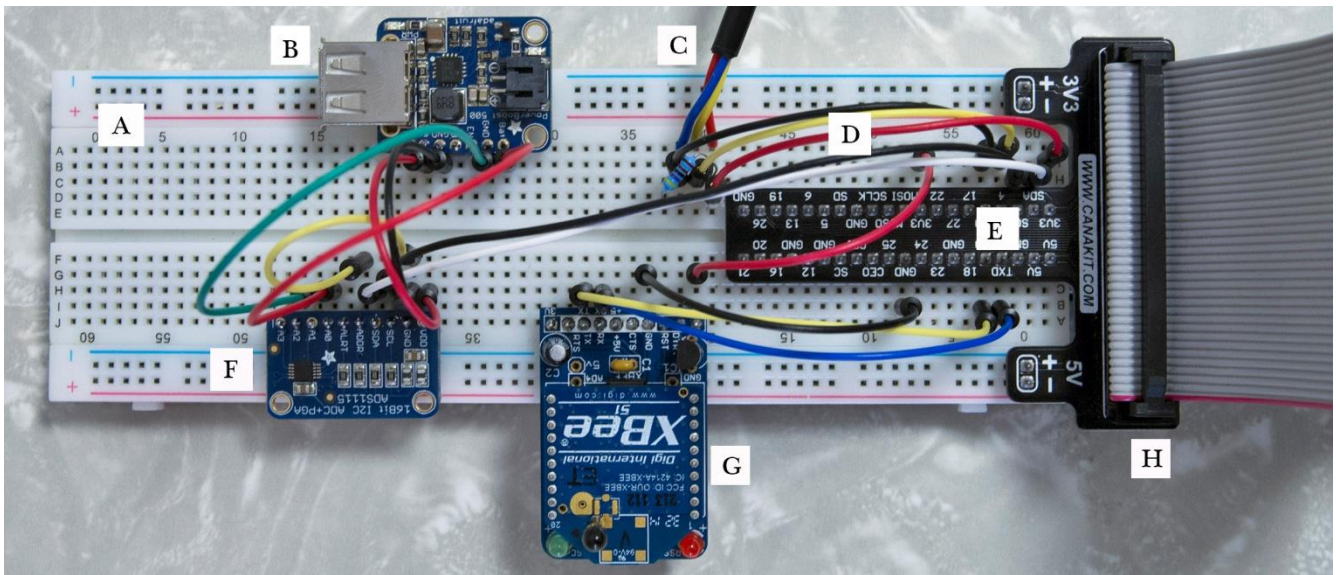


Figure 2.2: (A) Breadboard, (B) PowerBoost, (C) Temperature Sensor, (D) Male-Male Jumper Cables, (E) GPIO Breakout Board, (F) ADS1115, (G) XBee mounted on breakout board, (H) Ribbon Cable

2.2 Setting up the Raspberry Pi

The basic setup of the Raspberry Pi is very straightforward. For detailed instructions, refer to Section 1 of Appendix A. Once the operating system is loaded and configured, the Raspberry Pi functions just as a normal Linux computer. There are USB ports available for attaching a keyboard or mouse. There is an Ethernet port built in, or a Wi-Fi module can be attached, and there is an HDMI video output.

The Raspberry Pi also has a row of GPIO (general purpose input/output) pins. It is through these pins that we are able to connect other hardware components, such as sensors, to the Pi. The pin-out will be different depending on the model of the Pi, and aside from the input/output pins there will also be several pins for power and ground. While it is possible to connect some things directly into the GPIO pins, it is more common, and preferable, to work through a solder-less breadboard. A Raspberry Pi can be connected to a breadboard using a series of jumper cables but there are also specific pieces of

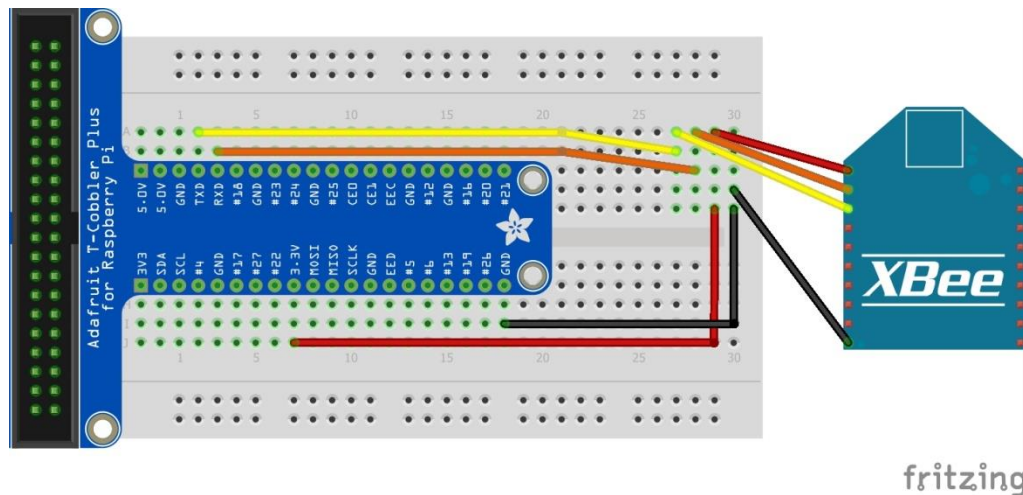


Figure 2.3: XBee wiring diagram

hardware known as breakout boards, which simplify the connection process. Using a breadboard and breakout board it becomes trivial to connect any number of components to each other and to the Raspberry Pi in an organized and contained package (Figure 2.2).

2.2.1 Configuring the XBee

The XBee modules need to be configured before being attached to the Pi. For instructions on configuring the XBee, refer to Section 8 of Appendix A. The XBee is wired to the Pi by connecting the data receiver on one (Rx) to the data transmit on the other (Tx) and vice versa (Figure 2.3). Aside from the various operating roles of routers, coordinators, and end points, there are also two different operating modes: AT and API. In the case of this project, we had one node set to coordinator API, and two nodes set to router API.

There are several key differences between AT and API mode. AT refers to “transparent” mode. In transparent mode, a router will send data to the coordinator, and the coordinator will read that data exactly as it was sent. However, the data will often be sent in several packets. In the case that there are

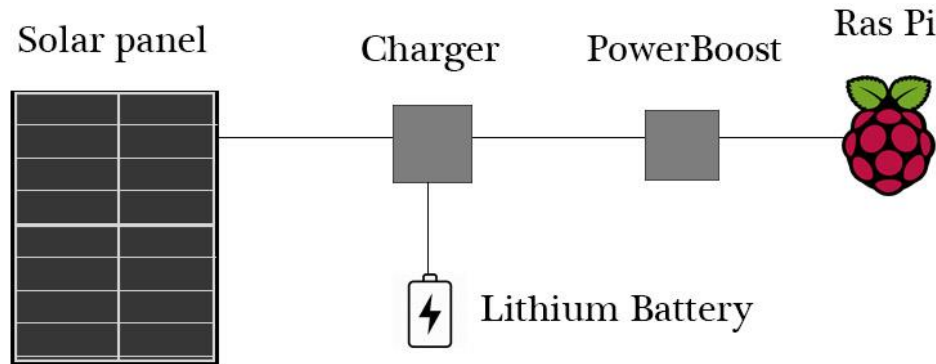


Figure 2.4: Solar array configuration; 6V 2W solar panel, 2500mAh battery

multiple nodes in the network, the coordinator might be receiving data from several nodes at once, which could end up with received messages being jumbled. For this reason, any network involving more than two nodes should be using API mode rather than transparent. API mode allows the addition of supplementary data to a packet, such as checksum and destination address. It also allows for the reprogramming of other nodes in the network, such as setting an end device into sleep mode.

2.3 Powering the Pi

In most cases of deployment, it is necessary to power the nodes by some means other than AC power. The most convenient way in the majority of situations will be solar power. In the example of data buoys out in a lake, the sensors would generally be in full sunlight, which is ideal for solar power. There are several options available for providing solar power to a Raspberry Pi. The hardware used in this project included a solar array, which was connected to a charging module, which was connected to both a rechargeable lithium battery as well as a USB power supply (Figure 2.4).

Sunlight is not always reliable as there may be a series of overcast days. One critical point about

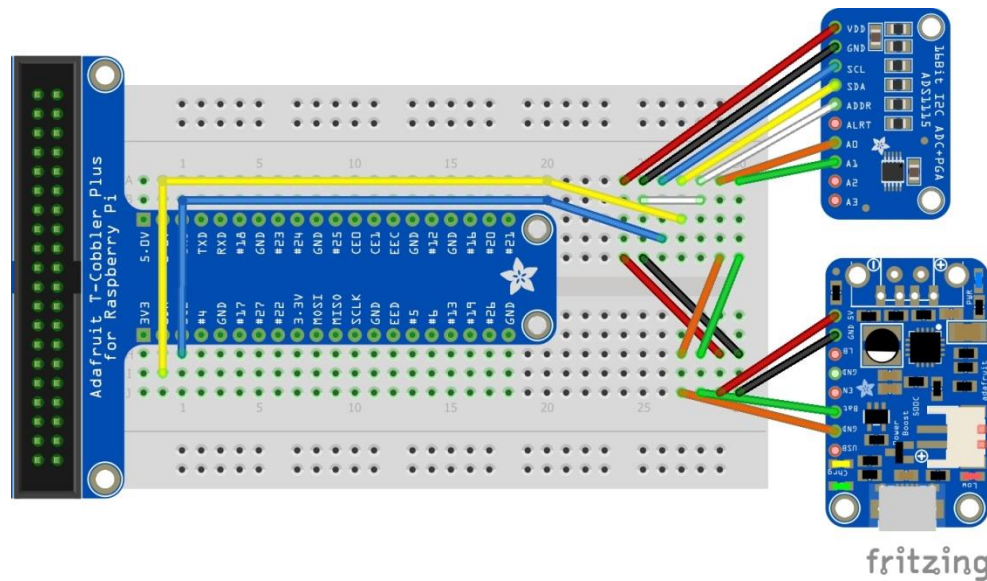


Figure 2.5: ADC wiring diagram

an XBee mesh network is the importance of the coordinator. If the coordinator goes offline then the entire network will fail. For this reason, if at all possible, it is preferable to keep the coordinator on AC power rather than solar. That being said, the Raspberry Pi consumes about 3.5W of power in normal operation and so it is entirely possible to gather enough power from the sun to power the Pi. However, power consumption should be taken into consideration and minimized wherever possible.

There are many battery sizes available depending on the power needs of the setup. The battery in this project was a 2500mAh lithium ion, but batteries at 4400 and 6600mAh are also commonly available. As mentioned previously, the XBee modules use very little power and if the network consists of several end devices it is possible to place them in sleep mode and reduce the power consumption even further. Another piece of hardware that can be useful to add to the sensors is a GPS. It is possible to simply note the GPS location of each sensor at deployment time, but it might be preferable for some to read the GPS information directly to the Pi and write it along with the sensor data. As a GPS can draw quite a bit of power, and because the sensors are not likely to be moving, it is

Table 2.4: Sensors and cost, available for Raspberry Pi

Sensor	Cost	Model	Notes
Accelerometer	\$5.00	Adafruit LIS3DH	
Camera	\$30.00	R Pi v2	Interfaces w/ Pi Csi port
Camera w/ motion detect	\$40.00	ViMicro	
Camera (waterproof)	\$55.00	Adafruit TTL JPEG	
Carbon Monoxide / Alcohol	\$15.00	Adafruit MiCS5524	
Color RGB	\$8.00	Adafruit TCS34725	
Distance/Range	\$15.00	Adafruit VL53L0X	
Force/Touch	\$8.00	Interlink 406	Analog signal
GPS	\$30.00	G-Star IV	USB interface
Light	\$7.00	Adafruit GUVA-S12SD	Analog signal
Liquid Flow Rate	\$10.00	Adafruit 828	Analog signal
Microphone	\$7.00	Adafruit SPH0645LM4H	
Motion	\$10.00	Adafruit PIR	
Orientation	\$35.00	Adafruit BNO055	
pH	\$150.00	Atlas Scientific	
Soil Temperature/Moisture	\$50.00	Sonbest SHT10	
Temperature (waterproof)	\$10.00	Adafruit DS18B20	
Temperature/Altitude/Pressure	\$10.00	Adafruit MPL3115A2	
Temperature/Humidity	\$15.00	Adafruit HTU21D	
Temperature/Humidity (waterproof)	\$30.00	Adafruit AM2315	
Temperature/Humidity/Pressure	\$20.00	Adafruit BME280	
UV Index / IR	\$10.00	Adafruit SI1145	
Wind Speed	\$45.00	Adafruit 1733	Analog signal

unnecessary to keep the GPS running constantly. Rather, the GPS can be programmed to take a reading once a day, or once a week, or at whatever interval seems appropriate.

In the case of a solar powered Pi, it can be very useful to know the current voltage of our battery. In the event that the power is getting low, this can give us the opportunity to respond, perhaps by disabling certain non-essential functions. The signal coming from the battery is an analog signal. However, the Pi is not able to read analog data, so an analog to digital convert (ADC) is required. The ADC took a dual channel reading from the PowerBoost, measuring the voltage difference between the

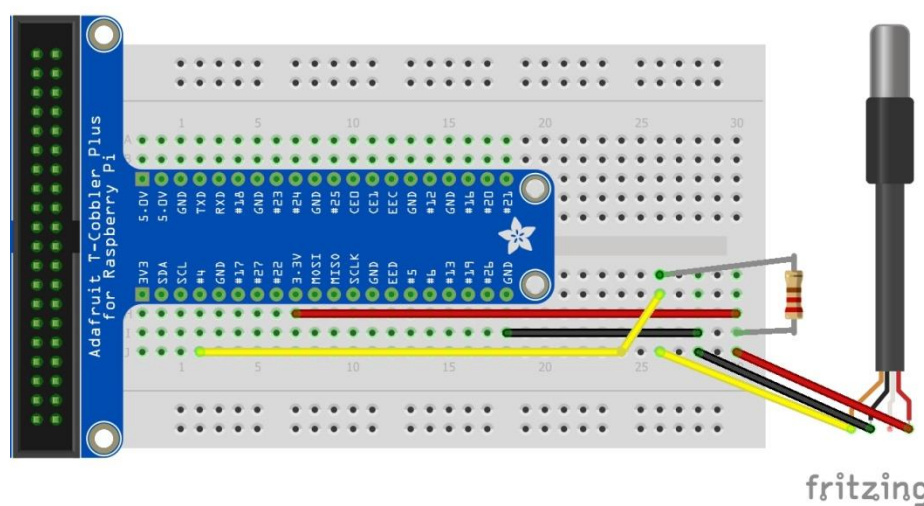


Figure 2.6: Temperature sensor wiring diagram

battery and ground signals and passing the reading to the Raspberry Pi (Figure 2.5).

2.4 Attaching sensors

There are any number of sensors available for the Raspberry Pi, allowing for the collection of many different kinds of data in many different environments. As the mechanisms required for measuring different variables can be quite different, there is also quite a bit of variation in the price of sensors. Table 2.4 lists some of the more commonly available sensors and their approximate cost. In some cases, the same sensors are available in either weatherproofed or non-weatherproofed versions. There are also several options available for weatherproof enclosures for the Raspberry Pi, although depending on the severity of the environment to which Pi will be subjected, it is just as common for many hobbyists to craft their own enclosures from common materials such as plastic containers, or even a water bottle.

There are also differences in how the sensors interface with the Pi. The sensors used in this project were very simple 3-wire temperature sensors. They simply require power and ground, and

connecting the signal wire to one of the Pi's digital inputs (Figure 2.6). These sensors are also capable of running in parallel allowing any number of these temperature sensors to be wired together. Because the unique serial number of each sensor is recorded with each reading, the Raspberry Pi is able to differentiate which signal is coming from which sensor.

While many sensors might interface easily with the Raspberry Pi, others might require the use of breakout boards. A breakout board is a piece of hardware that simplifies the process of connecting any device to a breadboard. Any piece of hardware will have some number of pins or wires that need to be connected to the Raspberry Pi. The spacing of the pins is not often such that the device can plug directly into the breadboard. The breakout board provides this correct spacing, keeping the setup clean and organized. Another difference among various sensors is how they transmit data. Some sensors only record analog data, and since the Pi only reads digital signals, an ADC is needed to convert the data signal into data that that Pi can read.

Chapter 3

Programming and setup

As the Raspberry Pi is a full Linux computer, the programming for the sensor network could have been done in any number of languages. The two choices that I considered were Python and Node.js. Python is a versatile general purpose programming language which includes an impressive standard library and a vast array of outside packages available. Node.js is a more recent language than Python but has already obtained a very strong following.

3.1 Node.js

Node.js is an open-source, event-driven JavaScript runtime environment. It has become very popular for web applications, as the event-driven interface is well suited to the user interaction of websites, and presumably because most web designers were already coding with JavaScript. It makes use of non-blocking asynchronous I/O and callback functions. When performing tasks that could take some time to complete, such as taking a reading from a sensor, it is necessary to employ asynchronous functionality. A callback function is a simple way to build asynchronous functionality into code. The callback is defined as an anonymous function and passed as a parameter, along with any additional parameters, when calling a function that includes a callback (Figure 3.1).

Like Python, Node.js has a large collection of open source libraries available. These packages can be added to Node projects using the node package manager, or npm. Every piece of hardware that I used for this project had at least one npm package available. The number and variety of npm packages greatly reduces the programming work involved when incorporating devices into a project.

```
// defining a function with a callback
function foo(callback) {
    // perform a task that takes some time
    callback(result); // pass the result to the callback function
}

var i;

// calling a function with a callback
foo(function(result) {
    // execute some code that is dependent on the value of result
    i = result;
    console.log(i); // output = result
});

console.log(i); // output = 'undefined' because code is executed
               // before the asynchronous function completes
```

Figure 3.1: A simple callback function

3.2 MySQL

There are at least two popular database options today, MySQL and MongoDB. MongoDB is what is known as a NoSQL database and the structure of the database is similar to what a programmer might think of as a dictionary, with key-value pairs. NoSQL databases may have an advantage over relational databases when complicated object relations are involved but since the database for this project was a simple one I chose to use MySQL.

MySQL is very easy to install and configure on a Raspberry Pi. See Section 5 of Appendix A for detailed setup instructions. There is an npm package available which makes interacting with a MySQL database very easy. For some example code, please refer to `write_db.js` in Appendix B.

The database structure is very important to consider for any project. The requirements for this project include being able to identify which node is sending the data, knowing the geospatial location of the node, knowing which sensor the data is coming from, knowing the voltage of the Raspberry Pi if it is solar-powered, and knowing some other general information about the node and the

Field	Type	Null	Key	Default	Extra
id	varchar(255)	NO		NULL	
node_id	int(11)	YES		NULL	
datetime	datetime	YES		NULL	
latitude	double	YES		NULL	
longitude	double	YES		NULL	
voltage	double	YES		NULL	
environment	varchar(255)	YES		NULL	
sensor_id	varchar(255)	YES		NULL	
depth_m	double	YES		NULL	
measurement_type	varchar(255)	YES		NULL	
measurement_units	varchar(255)	YES		NULL	
measurement_value	double	YES		NULL	

Figure 3.2: Database structure

sensor. Some of the database structure was modeled after the water quality data that is collected from the Lake Observer app, which includes the depth of the measurement if immersed in water, the measurement type, measurement units, and of course measurement value. This will allow us to use the same table structure regardless of what kind of sensor we are using. The full database structure is illustrated in Figure 3.2

3.3 Reading devices

The npm libraries enable us to easily interface with a huge number of devices. All of the device readings in this project are asynchronous, including temperature sensor, the analog-digital converter, and the GPS. In some cases there will be more than one library available, allowing us to choose the one that best suits our needs, and of course there is always the option to building a custom library.

The temperature sensor used in this project was a very simple 3-wire sensor, requiring minimal wiring to connect to the Pi and using very straightforward code to interact with and read from the

sensor. Refer to `read_temp.js` in Appendix B for some example code. The value returned from the temperature sensor looks like `[sensor_serial#, sensor_reading]`. The serial number was entered into the database as the `serial_id`, which in the case that we have multiple sensors attached in parallel to a Raspberry Pi we would know which reading came from which sensor.

The analog to digital converter used in this project, ADS1115, is a 4-channel converter, meaning that it can read from up to four different devices. It uses I2C, which is a communication protocol that allows one chip to talk to another. It can take both single-ended or differential readings. A single-ended reading will measure the voltage between the analog input channel and the analog ground. A differential reading uses two channels and measures the voltage between those two channels. For this particular project setup I took a differential reading between the battery and ground signals coming from the PowerBoost 500 module. The readings, taken in millivolts, were converted to volts. For some example code refer to `measure_voltage.js` in Appendix B.

Use of the GPS module with the Raspberry Pi requires the installation of `gpsd`. Section 7 of Appendix A outlines the installation and use of `gpsd`. By default, `gpsd` will start a GPS daemon automatically when the Pi starts up, which will query the GPS when it is available. Since we are trying to minimize power consumption it is better to disable the automatic daemon startup and instead start and stop the daemon from code. Refer to `gps_check.js` in Appendix B for example code implementing the GPS. The code will query the GPS and then write the latitude and longitude to our configuration file, which can then be read by the other modules.

3.4 Reading and writing files

It will often be useful to read or write files, and Node.js is capable of doing either one in a fairly straightforward manner. Reading files is done asynchronously. In this project I made use of two

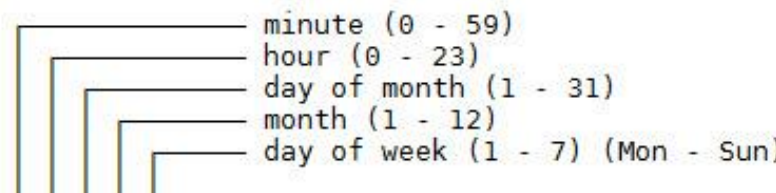
configuration files; `router.conf` and `coord.conf`. These configuration files contain information that is user-specific, or values that might vary from one node to another, such as database values, sensor values, and node values. Refer to `router.conf` in Appendix B for an example configuration file.

File writing is performed, as mentioned earlier, by the GPS, which writes the latitude and longitude to the configuration file. File writing was also employed the coordinator node, which backs up the database values and exports them to a csv file. That csv file is then sent to a remote server. Refer to `db_to_csv.js` in Appendix B for example code.

3.5 Sending data to a remote computer

The data recorded by the sensors is only useful if it can be accessed. In the case of an extended deployment, it is often preferable to have access to "live" data. This of course requires that the coordinator node, which contains the data from all the other nodes, has some form of access to an outside server. In the deployment of this project, the coordinator node was connected to the internet and so that data can be accessed at any point. When sending data from a local machine to a remote machine there are a number of options available, such as `sftp`, `scp`, and `rsync`.

Accessing a remote machine will requires logging to that machine with the necessary credentials. In order to do this in an automated way `ssh` keys will need to be used. An `ssh` key is a way for a remote machine to know that the machine which is asking permission for access does indeed have permission. This will allow us to send commands via `ssh` or `scp` and not be prompted to enter a password. Installation of `ssh` keys is outlined in Section 11 of Appendix A. Once the `ssh` keys are established we can then automate a connection to a remote host so that data is transferred at some specified time interval. That time interval is established using Cron jobs.



```

minute (0 - 59)
hour (0 - 23)
day of month (1 - 31)
month (1 - 12)
day of week (1 - 7) (Mon - Sun)

```

```

* * * * * program-to-execute command-to-execute

0 0 * * * : execute daily at midnight
0 0 1 1 * : execute every january first
@reboot   : execute when the machine boots up

// execute db_to_csv.js using node once every hour
0 * * * * /usr/local/bin/node /home/pi/dev/rasBuoy/node/coord/db_to_csv.js

```

Figure 3.3: Cron scripting

3.6 Cron jobs and bash scripting

One of the benefits of working with Raspberry Pi is having access to a full Linux operating system, which includes things such as Cron jobs and bash scripting. Cron is a scheduling agent that allows for the execution of any program or script at a particular time interval. Cron can specify the minutes, hours, days, or months. Figure 3.3 shows the form that a Cron job takes as well as a few examples. Section 10 of Appendix A lists all of the Cron jobs that were used for this project. The main router and coordinator scripts were both run using Cron.

Bash scripting is a way to encapsulate any Linux command into an executable program. A bash script was used to run a series of scp commands which would send several csv files to a remote system. The bash script was executed once every hour using a Cron job. The bash script used, `send_data.sh`, can be seen in Appendix B.

3.7 XBee

The code controlling the XBees was different depending on the role of the sensor node. For the coordinator node, the main Node.js file, coordinator.js, is run at boot time. The program will wait to receive data from other nodes. Any data received will be entered into the database. The database is then converted to several csv files every hour via Cron jobs and sent to a remote machine. The csv files are then read by that machine and displayed in a web portal.

The main router file, router.js, is also run at boot time. It will also listen for data. Any data received is checked against the database to see if the data has already been received. In a network with multiple routers connected and passing on data it is possible for the same piece of data to wind up at the same router more than once. If the data is new it will be saved to the database and then sent out to the other nodes. Aside from listening for data, the router will also record data every fifteen minutes, save it to the database and send it to the other nodes. It is also possible for a router to be programmed to act as an end device, only recording and sending out data, but not receiving data.

Chapter 4

Deployment and results

Whenever writing code, it is imperative that the code be thoroughly tested somewhere along the process. When assembling hardware, we must similarly test the functionality of the setup. In the case that we are testing both the hardware and software there are additional challenges. Because everything must interact and function together, if things don't work as expected, it is not always readily apparent if the problem is hardware or software related. This project is an amalgamation of many different pieces and a successful deployment of the project must make sure that all those pieces fit together properly.

4.1 Initial deployment setup

The initial deployment of the framework was a very simple proof of concept deployment. There were several goals of this initial deployment. The first goals were related to the programming. The intended function of the code was to read data from the sensors, write the data to a database, and send the data to the other nodes in the network. The second set of goals were related to the functionality of the hardware itself. The various pieces of hardware each had separate roles to play; reading environmental data, transmitting data between nodes, measuring voltage, and sustaining power.

4.1.1 Nodes and roles

The full setup included three Raspberry Pi / XBee nodes; one coordinator, one router, and one end device. The coordinator was hooked up to AC power and also connected to the internet via an Ethernet cable. The router node was being powered from a solar array and set read the temperature from a sensor every fifteen minutes. It was also hooked up to ADC and set to read the voltage from the

battery at the same time that it reads the temperature. The end device was also on solar power and set to read the temperature from a sensor every fifteen minutes. Both the router and the end device were positioned in full sun in a south facing window. The solar array does not have optimal function behind a window but an important part of testing the framework is to test it in less than optimal conditions, as it should be expected to sustain power on cloudy days as well as overnight when it will be receiving no solar charge at all.

4.1.2 Data flow

The data is read from the sensors of the router and end device, with each node recording the sensor data to their own local database. After writing the data it is broadcast to all other nodes in range within the network. In this case, all nodes were within range of each other. The router will receive incoming data, write that data to its database, and broadcast that data again out to all nodes in range. Whenever data is received by a router or coordinator it will be checked against the data already within its database. New data will be written to the database, and in the case of the router node, the data will be broadcast to the other nodes.

The coordinator node records all received data in its database. Barring any kind of failure in the routing, all data in the network should ultimately make its way to the coordinator so that its database contains all information recorded from all other nodes. Selected data is then read from the database and written to csv files every thirty minutes. In this deployment there were four separate csv files; one file for the temperature readings from Node 0, one for the temperature readings from Node 1, one for the voltage readings from Node 0, and one for the most recent temperature and voltage readings from both nodes. All of these csv files are then sent every thirty minutes to a remote server.

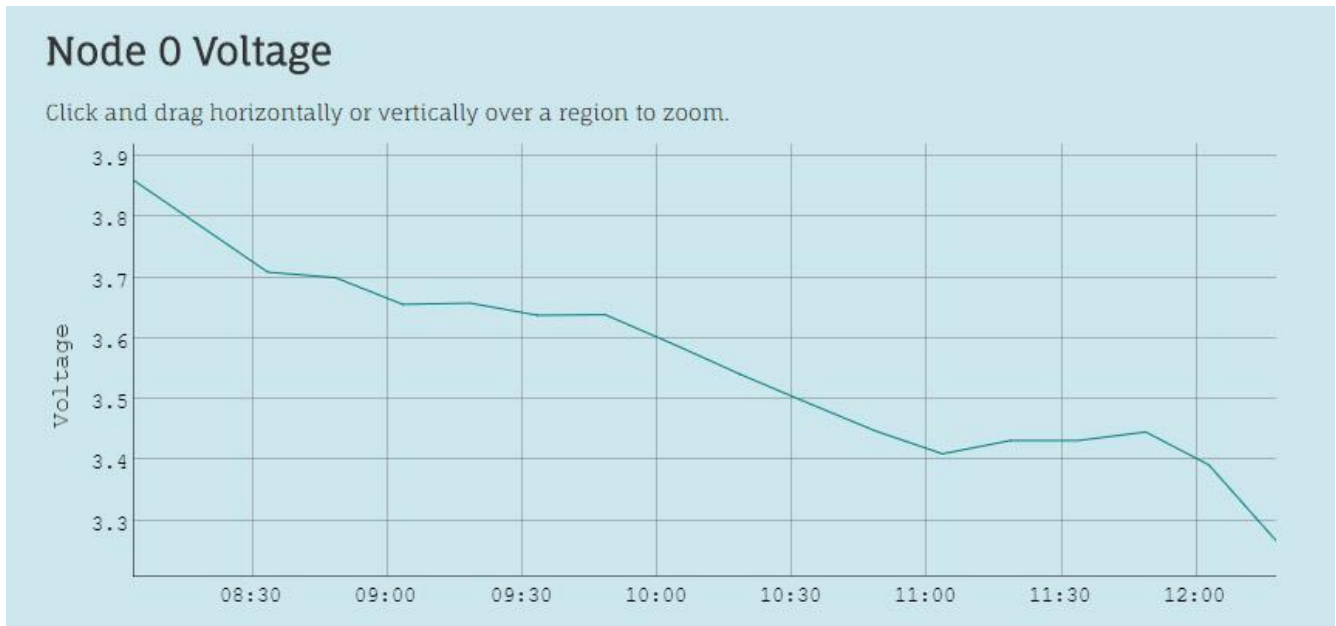


Figure 4.1: Graphical display of voltage data

4.1.3 Data view

After the data is sent to the remote server it is then published to a web site. With the data being refreshed every thirty minutes, the result is effectively having a live display of the sensors. This is one of the benefits of having a coordinator node connected to the internet. While it is not a necessity to have a live feed of the sensor data, it is certainly beneficial in many cases. The csv files that are received by the web server are displayed graphically using a simple JavaScript based web widget, which also allow for zooming into specific time periods (Figure 4.1). Aside from the graphical displays there is also a tabular view showing all the readings available from the various sensors as well as the timestamp for the data (Figure 4.2).

Latest Readings			
Node ID	Date & Time	Temperature (C)	Voltage (V)
0	Fri Mar 24 2017 15:35:01 GMT+0000 (UTC)	24.1	3.267
1	Fri Mar 24 2017 15:47:47 GMT+0000 (UTC)	23.9	

Figure 4.2: Latest readings from all available nodes

4.2 Results and figures

While this is a sensor network, the actual sensor data in the initial deployment is fairly inconsequential to the effectiveness of the framework. It is enough to say that the sensor data was correctly recorded. The data that is very significant to the functionality of the framework are the voltage readings. One of the premises of the framework is that it should be able to operate, for a theoretically indefinite time, using only solar power. Unfortunately, the results of the initial deployment showed that the selection of hardware for the solar array was not sufficient to power the sensors nodes.

The full voltage readings from the deployment can be seen in Table 4.1. The initial charge on the battery was about 3.86 volts, which would have been fully charged or fairly close to fully charged. However, only four hours and fifteen minutes elapsed from the time of the first transmitted reading to the time of the last transmitted reading. It should be noted that while the router node was transmitting the sensor data that it was collecting, which was a total of seventeen readings, it was also receiving and routing an equal number of readings from the end device node. This initial deployment helped to highlight several potential problems, which might have several possible solutions.

Table 4.1: Voltage readings from initial deployment

Date Time	Voltage
Fri Mar 24 2017 12:03:34 GMT+0000 (UTC)	3.861
Fri Mar 24 2017 12:33:36 GMT+0000 (UTC)	3.709
Fri Mar 24 2017 12:48:37 GMT+0000 (UTC)	3.7
Fri Mar 24 2017 13:03:38 GMT+0000 (UTC)	3.656
Fri Mar 24 2017 13:18:39 GMT+0000 (UTC)	3.658
Fri Mar 24 2017 13:33:40 GMT+0000 (UTC)	3.638
Fri Mar 24 2017 13:48:41 GMT+0000 (UTC)	3.639
Fri Mar 24 2017 14:03:42 GMT+0000 (UTC)	3.591
Fri Mar 24 2017 14:18:43 GMT+0000 (UTC)	3.541
Fri Mar 24 2017 14:33:43 GMT+0000 (UTC)	3.494
Fri Mar 24 2017 14:48:44 GMT+0000 (UTC)	3.448
Fri Mar 24 2017 15:03:45 GMT+0000 (UTC)	3.41
Fri Mar 24 2017 15:18:46 GMT+0000 (UTC)	3.432
Fri Mar 24 2017 15:33:47 GMT+0000 (UTC)	3.432
Fri Mar 24 2017 15:48:48 GMT+0000 (UTC)	3.446
Fri Mar 24 2017 16:03:48 GMT+0000 (UTC)	3.392
Fri Mar 24 2017 16:18:49 GMT+0000 (UTC)	3.267

4.3 Successes and failures

While the problem with the solar power was a major issue, which prevented the framework from functioning in its entirety, the rest of the hardware and programming seemed to function without any major problems. The data from read from the sensors, written to the databases, and transmitted to the coordinator in fifteen minute increments, without any loss of data. The voltage readings from the ADC gave us a very clear picture of how quickly the battery was drained. They also gave us an idea of minimum power requirements for the Raspberry Pi. This opens up the possibility of adding additional programming into the framework that might respond to changes in voltage. However, before we know what responses might be appropriate it would be helpful to know which aspects of the framework are drawing the most power, which would require more testing of the framework under various conditions.

4.4 Further testing

With only a single deployment it is difficult to say exactly what the main source of the problem might be. The easiest solution would be to find the largest possible battery and a larger solar array and run the deployment again. While this might correct the problem of running out of power, it does not necessarily give us a clue as to where the main source of the power drain is coming from. A better solution might be to find a way to make the framework function on the current battery and solar array and then upgrade the power setup to ensure that things will run more effectively.

There are several questions that would be useful to answer. How much more power does the Raspberry Pi consume when running with the XBee module? How much does it consume running simply by itself? How much power could be saved when using the XBee as an end device compared to a router? And how much power do the additional peripherals, such as the sensor and the ADC consume? In order to answer these questions we would need to run the deployment several more times with several different hardware configurations. Of course, one of the main benefits of a DIY framework is the freedom to easily make modifications to the system and add or subtract things from the setup to suit a project's particular needs or limitations.

4.4.1 Additional deployments

Two additional deployments were made after the first in an attempt to address some of the questions regarding power drain from the various components, as well as the assigned roles of the components (i.e. router vs. end device). The voltage drain from these various deployments could then be compared to get an idea of what aspects might be changed in order to minimize power usage. It should be noted that the differences in voltage drain, while readily apparent, cannot be used to judge

Table 4.2: Voltage drain from various deployments

Deployment	Start voltage	End voltage	Time elapsed	Voltage drain
XBee (w/ routing), temperature sensor, voltage reader	3.861	3.267	4.25 hours	0.140 v/hr
XBee (w/ out routing), temperature sensor, voltage reader	3.55	3.276	2.75 hours	0.100 v/hr
Voltage reader only	3.737	3.331	7.00 hours	0.058 v/hr

the power drain with certainty since there are other factors that might be affecting the charge of the device, namely the amount of sunshine hitting the solar panel.

The only modification made to the second deployment was to modify the code of the router so that it was no longer routing data from the end device to the coordinator. This effectively turned the router into an end device, although it did not take advantage of the main benefit of an end device, which is the potential for sleep mode. The third deployment eliminated both the XBee module and the temperature sensor. This was meant to provide a baseline for the power requirements of the Raspberry Pi on its own. As might be expected, the Raspberry Pi ran for significantly longer without any other attached hardware (Table 4.2).

Chapter 5

Future enhancements and applications

The sensors have been deployed and the data collected, but this is certainly not the end of the project but only the first stage of this project. There are many improvements that need to be made, and many others still that could make the project better and more effective. The hope is that I have laid the groundwork, both for myself and others. The initial deployment works only as a proof-of-concept, to show that the framework has potential to be very useful in the future. I will continue working on the project in the days and weeks and years to come and make improvements where needed.

5.1 As an open source framework

The best way to help a project develop is to keep it publicly available and open source. This allows anyone to deploy the project and make their own modifications, or at the very least suggest some improvements that can be made. The project is currently being maintained as a public GitHub repository at <https://github.com/mForcella/rasBuoy>. This makes it publicly available but does not guarantee that it will be found or used. The challenge one faces is how to get a project to the correct audience. Though, before attempting to reach out to a particular audience, it is important to consider who the project might be most useful to.

There are many potential users for this framework, including students and academics, researchers, or simply hobbyists or enthusiasts. Reaching hobbyists might be as simple as writing a blog post and waiting for someone in need to find it. I certainly read through a fair number of blog posts as I was searching for answers to questions that came up while working on this project. However, posting and waiting offers little guarantee that anyone will find or use the framework. Bringing it to the

attention of academics could mean reaching out to professors, which in the case of this project might be biology or computer science professors. There is also the possibility of finding a journal, online or print, that might be willing to publish the project. But before this can happen it would be wise to make sure that the framework is functioning without any major issues.

5.2 Needed improvements

As this project is still in its infancy it is no surprise that there are still several issues that need to be worked out. One cannot generally expect a project to run smoothly from the onset. As the use of a project increases, the potential for uncovering problems also increases. This is an essential part to the development of any project, and as such, the more thoroughly a project can be tested, and the more issues that can be revealed, the better.

5.2.1 Power consumption

The main issue faced was that of power consumption. Although the Raspberry Pi is a low-power machine the power setup employed was not sufficient. The solar panel that was used was only a 2W panel, while the Raspberry Pi runs at about 3.5W. Given these numbers it seems apparent that larger solar panel is necessary to run the Pi indefinitely. The question then becomes, what size solar panel would be sufficient? One might think, the bigger the better, but aside from the increasing cost of larger panels, there is also an increase in size and weight that also needs to be considered. A summary of solar panel sizes along with their cost is available in Table 5.1.

The other change that we might consider would be in the code. There might be something in the code that is causing the hardware to consume power unnecessarily. There are also hardware settings, specifically within the XBee, that could be modified. The XBee modules have many configuration

Table 5.1: Cost, size and weight of solar components

Item	Cost	Size	Weight
Solar Panel 2W	\$29.00	4.4" x 5.4"	90 g
Solar Panel 3.5W	\$39.00	4.4" x 8.3"	140 g
Solar Panel 6W	\$59.00	6.9" x 8.7"	225 g
Solar Panel 9W	\$79.00	8.8" x 10"	309 g
Lithium Battery 2500mAh	\$15.00	2" x 2.55"	52 g
Lithium Battery 6600mAh	\$30.00	2.1" x 2.7"	155 g

settings that I have yet to experiment with, including different power levels and sleep settings. The highest power level available was used in the deployment to ensure sufficient range, but perhaps it was more than necessary.

There is also additional hardware that can be added to the framework. Specifically, there are devices available for cycling power. These devices are able to turn power on and off. If, hypothetically, the Raspberry Pi was programmed to turn on for five minutes and then power off for five minutes, this would effectively double the battery life of the Pi. This would require much more complex programming to deal with specific nodes becoming unavailable, but would

5.2.2 Messenger queues

As is often the case with hardware, there are several features available on the XBee modules that were not used. There are in fact entire books written on the subject of ZigBee mesh networks [16]. Operating in API mode, the XBees send additional information along with each data frame. This includes the source address of the received packet, success and failure status of the packet, and the signal strength of the packet. Having this information allows the network to operate in the same fashion as a network running on messenger queues. The idea of messenger queues is that there will be a

publisher of some message, and one or more subscribers to that message. The messenger queue is there to guarantee the delivery of that message. No such guarantee was built into this framework, but having the API frame data would allow us to program in that guarantee.

In the very small network that was used for the deployment in this project there was not any noticeably failure in sending or receiving data. However, in a larger network with multiple routers and a potentially greater distance between nodes, there is a much greater chance for communication failure. With that in mind, before attempting to deploy a larger network, it would be imperative to build in some method to verify and guarantee that a message being sent will reach its destination. As well as guaranteeing the message deliver, it is also important create methods that will reduce or prevent sending unnecessary or redundant data, as this will only increase the chances of failure and error.

5.2.3 Error checking

Although no serious errors were experienced during deployment, we must keep in mind that this was a very small and contained network. As the size of a network increases, the chances of error and failure also increase. It can be difficult to anticipate the types of errors that the network might encounter and so the only way to reliably find the errors would be to deploy the project until those errors are encountered. The best way to coax these errors out would be to intentionally deploy the project in adverse conditions. By creating a sort of worst case scenario we increase the chances that our project will not fail in these or better conditions.

5.3 Potential enhancements

While some improvements are necessary to ensure that the framework functions correctly in any and all conditions, there is also room for additional enhancements that may not be necessary but

will improve the functionality of the framework. There might also be the need to customize the framework in some fashion to fit the needs of a particular project.

5.3.1 Data download

For the deployment of this project, the coordinator node was connected to the internet via Ethernet. This is certainly not the case in all deployments. Most environments that we would wish to collect data from are isolated in some way. In many cases, environmental sensors will be connected to an outside server via cell signal. This is very reliable but it can be costly, as aside from the equipment it generally involves a monthly fee for the service. This may not be feasible for many DIY projects. Barring access to Ethernet, Wi-Fi, or cell signal, the only other option might be to forego having live data access and instead rely on manually collecting the data.

When talking about manual data collection, this does not necessarily mean that we need to take the sensor node, hook it up to a computer, and download the data. There are wireless options available for accessing a Raspberry Pi. One option is Bluetooth, which has a range of about 100 meters. There are Bluetooth devices that can be attached to a Raspberry Pi, although attaching more hardware also means drawing more power. Assuming that the Pi can handle the additional power requirements, the data could then be downloaded, from some distance, by a computer, cell phone, or any device that is Bluetooth enabled.

Another option available involves a messenger queue service called MQTT. This is a very lightweight service specifically designed to run on low-power devices. MQTT can also run on a laptop or cell phone. With an MQTT server running on the coordinator node, we could theoretically approach our sensors with something like a smartphone, running an MQTT client, and functioning as a wireless hotspot. The Pi would then connect to the hotspot and transfer any queued data.

5.3.2 *Integration with public database*

I spoke briefly about the benefits and need for public data access (Section 1.1) and this should certainly be a consideration for this project. It is all very well and good to have our sensors collecting data but what happens to the data afterwards is even more important. Thinking in terms of citizen science, let us picture the framework in the hands of an ecological enthusiast, or a retired K12 teacher. This well meaning individual has a pond or stream on their property that they wish to monitor. With their sensors collecting data they can now get a better picture the health of their pond or stream. This pond or stream may well be a part of a much bigger watershed, and if someone was interested in studying that watershed they might find this person's data very useful. But if they wish for their data to be put to a larger purpose it needs to be made available to the right person. This is a key feature of any good tool for citizen scientists.

One way to do this would be to create a public database that users of the framework could use with their projects. This database could also be linked to a web portal that would allow users to view their own data and the data of others. This is very much along the lines of what Lake Observer and GLEON are trying to do. Making it easy for groups or individual to share the data that they are gathering with a collective whole. This is beneficial to the people collecting the data, as it makes their data more meaningful, and of course it is useful to whatever groups or organizations that might be interested in using the data.

5.4 **Future applications**

The great hope of this project is that this framework will someday be put to use in a meaningful way. There are many potentially useful applications for low-cost deployments of environmental sensors, both broad and local. And as can often be the case, something that we perceive as a local issue

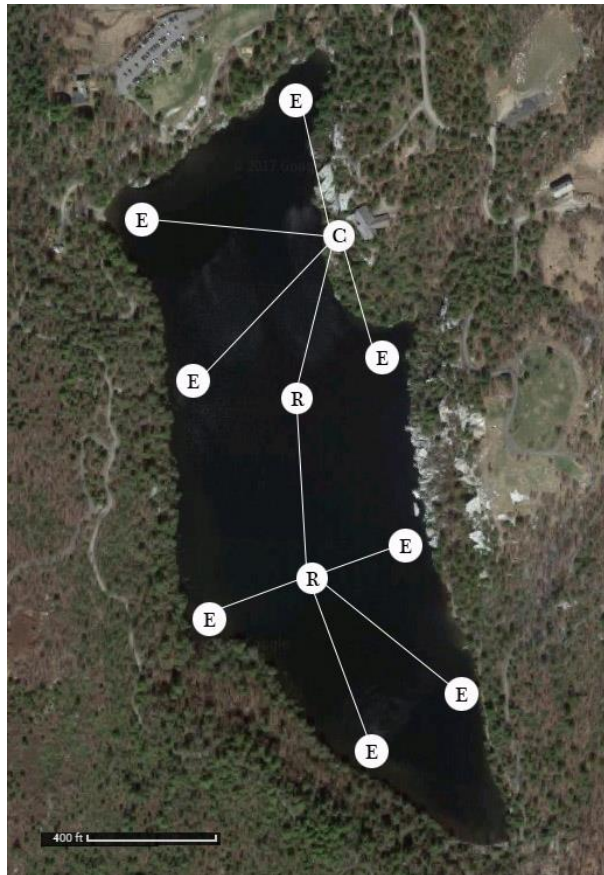


Figure 5.1: Hypothetical mesh network on Lake Minnewaska

is actually just a smaller part of a much larger issue.

5.4.1 Local issues

Here in the Hudson Valley there is ongoing environmental research in many areas. One of the more prominent issues is the state of our watersheds. One in particular which has received some attention lately is the Wallkill river. The river has experienced problems with algal blooms and fecal contamination for some time now.

One of the organizations that is monitoring the river is Riverkeeper, which is a non-profit organization dedicated to protecting the Hudson River and its tributaries. They have a fairly extensive

citizen monitoring program which regularly goes out and collects data from various points in the watershed. The public is then able to see through the Riverkeeper website when these tests come back with high levels of contaminants and a "beach advisory" has been declared. Adding sensors to certain areas would make monitoring easier and enable more frequent updates on water conditions.

Another very important local issue involves several bodies of water, collectively referred to as the Sky Lakes. These are several lakes on the Shawangunk Ridge with very small watersheds relative to their lake size. Many of these lakes turned acidic several decades ago and became relatively lifeless. However, one of these lakes, Lake Minnewaska, has become much more alkaline in recent years, with fish beginning to return to the waters [17]. The reason for this change in pH however, is not yet known. There are several theories, but none have been confirmed. Establishing a mesh network on the lake, with perhaps a dozen sensors collecting pH data at various locations and depths, could provide an extraordinary amount of data (Figure 5.1). It is just this kind of data which might offer us some clues regarding the changing pH.

5.5 Final thoughts

This framework, while not yet perfected, has certainly demonstrated the potential to fill a neglected niche. The initial testing discovered some flaws and points of improvement. Further testing and development would help to bring the scope and shape of the project into more clear focus. The potential applications are many and the hope for the future is that the project will continue to evolve and be put to use on a larger scale and to a purposeful end.

Appendix A

README file and setup instructions

Note: these instructions are written for a Raspberry Pi model B+ running Raspbian Jessie Lite and instructions may vary depending on your model and operating system.

Section 1: Setting up your Raspberry Pi

Hardware used:

Raspberry Pi (used model B+ in my setup)

SD card (4GB minimum, 8GB recommended)

USB keyboard

USB mouse (optional, if using an OS with a graphical user interface)

Monitor or other display (must be able to accept HDMI output from the RPi. HDMI to serial converters are available to interface with most monitors)

Prepare your SD card.

You will need to download an operating system and burn it to your SD card. Various operating systems are available, some with a graphical user interface and some without. I used the Raspbian-Jessie-Lite OS. Adafruit has some tutorials to help you with the setup.

> <https://learn.adafruit.com/adafruit-raspberry-pi-lesson-1-preparing-and-sd-card-for-your-raspberry-pi>

Note: Not mentioned in the Adafruit tutorial, Etcher is a very useful cross-platform (Windows, OSX, and Linux) program for flashing images to SD cards. It allows you to flash images without unzipping the file and also includes a validation step to ensure that the file was written correctly. This can be very useful, as I have experienced OS errors due to bad image copies.

After you've installed your OS and established an internet connection (your Pi should connect to the internet automatically via wifi or Ethernet) it is a good idea to run 'sudo apt-get update' and 'sudo apt-get upgrade' to make sure you have the latest updates to the OS.

Section 2: Enabling SSH

If you want to be able to log into your Pi from another computer you will need to enable SSH. In order to enable SSH on your Pi you will need to run 'sudo raspi-config' and under 'advanced options' find and enable SSH.

You will also want a static IP address on your Pi. If your internet router lets you setup a DHCP reservation for a particular device this is probably ideal as it will not require you to make any configuration changes on your Pi. You can also set a static IP address on your Pi by modifying the /etc/dhcpd.conf file, but the configuration will vary depending on your internet setup and some configurations may be more complicated than others. My own configuration is below.

```
> interface eth0
> static ip_address=192.168.1.xxx/24
> static routers=192.168.1.1
> static domain_name_servers=192.168.1.1
```

If allowing SSH from an outside computer it's probably a good idea to change the default password of your Pi. Do this by typing the 'passwd' command.

Section 3: Installing git

Git is a tool for sharing and collaborating on files and projects via the GitHub website. If you're interested in learning more about git, Udacity has a free course available.

> <https://www.udacity.com/course/how-to-use-git-and-github--ud775>

Install git by running the following command:

> `sudo apt-get install git`

To use git with GitHub you will need to create a GitHub account and configure git with your username and email. Basic configuration instructions can be found here:

> <https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup>

Section 4: Installing Node.js

Node.js is a JavaScript platform that we will use to run various scripts. The version available with many Linux distributions is outdated, but the newest version can be obtained via GitHub at <https://github.com/audstanley/NodeJs-Raspberry-Pi>. Install with the following:

```
> git clone https://github.com/audstanley/NodeJs-Raspberry-Pi
> cd NodeJs-Raspberry-Pi
> chmod +x Install-Node.sh
> sudo ./Install-Node.sh
> cd .. && rm -R -f NodeJs-Raspberry-Pi/
> node -v
```

This will also install npm, which is the Node Package Manager and can be used to install various packages.

Section 5: Installing MySQL and setting up database

```
> sudo apt-get update
> sudo apt-get install mysql-server && sudo apt-get install mysql-client
Enter root password when prompted. Make note of password.
```

Log in to MySQL server with command:

```
> mysql -u root -p
```

Create a database:

```
> CREATE DATABASE rasBuoy;
```

Select your database:

```
> use rasBuoy;
```

Create tables:

```
> CREATE TABLE SensorData (id VARCHAR(255) NOT NULL PRIMARY KEY, node_id INT,
datetime DATETIME, latitude DOUBLE, longitude DOUBLE, voltage DOUBLE, environment
VARCHAR(255), sensor_id VARCHAR(255), depth_m DOUBLE, measurement_type
VARCHAR(255), measurement_units VARCHAR(255), measurement_value DOUBLE);
```

Section 6: Configuring the temperature sensor

Hardware used:

DS18B20 waterproof temperature sensor

Male-male jumpers

Breadboard

There are any number of sensors available that can be attached to a Pi. One of the cheaper and more readily available is the DS18B20 temperature sensor, which we will use to demonstrate the process of reading data from the Pi's GPIO (general purpose input/output) port. A simple tutorial is available from Adafruit:

<https://learn.adafruit.com/adafruits-raspberry-pi-lesson-11-ds18b20-temperature-sensing>

After connecting the temperature sensor to your Pi you will need to edit one of the Pi's config files. Open the file with the following command:

```
> sudo nano /boot/config.txt
```

Adding the following line to the bottom of the file:

```
> dtoverlay=w1-gpio,gpiopin=4
```

Run 'sudo reboot' to restart your Pi. After the Pi starts up, load the drivers for the sensor with the following commands:

```
> sudo modprobe w1-gpio
```

```
> sudo modprobe w1-therm
```

To confirm that the temperature sensor has been correctly loaded, navigate to the directory, `cd /sys/bus/w1/devices/`, and type 'ls' to see a list of files. You should see a directory that begins with 28. This number is the serial number of your sensor.

If you'd like these drivers to load automatically when you boot up your Pi you can modify a config file with the following command:

```
> sudo nano /etc/modules
```

and add the following lines to the file:

```
> w1-gpio
```

```
> w1-therm
```

Section 7: Configuring the GPS

Hardware used:

G-Star IV USB GPS module

Install the gpsd software:

```
> sudo apt-get install gpsd gpsd-clients
```

The gpsd client is configured to start automatically at boot time but this can interfere with creating manual instances of the gpsd daemon.

Open the following file:

```
> sudo nano /etc/default/gpsd
```

and change the value of `START_DAEMON` from 'true' to 'false'. Now run the following two commands:

```
> sudo systemctl stop gpsd.socket  
> sudo systemctl disable gpsd.socket
```

Note: If you'd ever like to re-enable the autoloading of your gpsd daemon you can run, 'sudo systemctl enable gpsd.socket' and 'sudo systemctl start gpsd.socket'.

Now restart your Pi. After reboot you will need to figure out which port your GPS device is using. If you only have a single USB device on your Pi the port value would be `ttyUSB0` but if you have multiple devices this might not be the case. Unplug the GPS from the Pi and then plug it back in. Now type the following:

```
> tail /var/log/syslog
```

This will display the last few lines from your system log file. One of the lines should read "such and such device now attached to `ttyUSBx`". Make a note of which port the device is attached to and run the following command to start the gpsd daemon (replacing '`ttyUSBx`' with whatever port your GPS is using):

```
> sudo gpsd /dev/ttyUSBx -F /var/run/gpsd.sock
```

Now run '`cgps -s`'. This should bring up a display of your GPS. If all goes well it should acquire some satellites and start displaying information. If you get a 'gps timeout' error you might need to further configure your GPS. Configuration may vary depending on your hardware and operating system. You might find more detailed instructions here:

```
> https://blog.retep.org/2012/06/18/getting-gps-to-work-on-a-raspberry-pi/
```

Section 8: Configuring the XBee Modules

The XBee modules are configured using the X-CTU software. The software is available for free for any operating system. In order to connect your XBee to your computer you will need a USB interface, the most commonly used of which is called the XBee Explorer. When you add a new XBee S2 module to X-CTU it's "Function" will probably be set to ZigBee Router AT. Set the firmware function to whatever mode you want, coordinator, router, or end point, and the modules should be ready to deploy. If your network has more than two nodes you will need to use API mode rather than AT.

Other than the firmware version the only other option that needs to be set is the PAN ID value. This is the network identifier, and every module must have the same PAN ID in order to communicate with each other.

Once the XBee modules are set to the appropriate mode they are ready to be connected to the Pi. The most basic way to connect the XBee to the Pi would be to wire it directly to the breadboard using jumper cables. This will require making the following connections: XBee 3.3V pin the Pi 3.3V pin, XBee ground to Pi ground, XBee Tx (data out) to Pi Rx (data in), and XBee Rx to Pi Tx. More commonly though you would use an adapter boards. Adapter boards might have serial or USB interfaces and they add features such as LED indicators for power and activity, as well as facilitating swapping modules and connecting them to a computer for configuration. One of the boards which we have already discussed is the XBee Explorer, which will also allow you the connect the XBee to the Pi via USB rather than GPIO.

If you're running the XBee through a USB port, no further setup is necessary. However, if you're using the GPIO, you will need to open the serial ports. You will need to do several things in order to do this.

Firstly, make sure that the username (default is pi) is a member of the dialout group.

```
> sudo usermod -a -G dialout pi
```

Second, open up the /boot/cmdline.txt file and remove any reference to serial0. For example, "dwc_otg.lpm_enable=0 console=serial0,115200 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 rootwait"

would become

```
"dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 rootwait"
```

Third, open /boot/cmdline.txt and add the following line to the end of the file:

```
> enable_uart=1
```


Section 9: Configuring the ADC Module

Before you are able to connect the ADC to the Pi you will need to enable I2C, which is a communication protocol that allows chips to talk to each other. First install the I2C utility.

```
> sudo apt-get install -y i2c-tools
```

Enable I2C in the Raspberry Pi config by running `sudo raspi-config`. Depending on your version, the option to enable I2C will either be under Interfacing Options or Advanced Options.

Next, open the `/boot/config.txt` file and add the following lines:

```
> dtparam=i2c1=on
```

```
> dtparam=i2c_arm=on
```

Reboot the Pi. You can check to see if your ADC is detected using the following command:

```
> sudo i2cdetect -y 1
```

It should show a list of I2C addresses that are currently in use.

Section 10: Scheduling Cron tasks

Cron is a program built into Linux that allows you to automate tasks to run at a particular interval or time. For the router node, we will run router.js every fifteen minutes. To create a new crontab enter:

```
> crontab -e
```

When prompted for which editor to use, simply press enter to use nano.

For the router nodes, add the following line to the end of the script.

```
> @reboot /usr/local/bin/node /home/pi/dev/rasBuoy/node/router/router.js
```

This will run the script on boot.

On the coordinator add the following:

```
> @reboot /usr/local/bin/node /home/pi/dev/rasBuoy/node/coord/coordinator.js  
> 0 * * * * /usr/local/bin/node /home/pi/dev/rasBuoy/node/coord/db_to_csv.js  
> 0 * * * * /bin/bash /home/pi/dev/rasBuoy/node/coord/send_data.sh
```

The first '0' means, 'at 0 minutes on the hour'. The other characters indicate every hour, every day of the week, every month, every year.

Section 11: Sending files via scp

scp is a very simple way to send small files between systems. I will use scp to send a csv file of sensor data to a remote server that hosts a website that will display the csv data. The scp commands will be issued via cron scheduled bash scripts. In order to use the scp command from a bash script we will need to establish ssh keys in order to connect to the server without a password.

To create the ssh keys enter the following command on the rpi.

```
> ssh-keygen
```

You will be prompted to select a directory to store the keys. Unless you want them in a specific location, you can simply accept the default. Now you must copy the keys to the remote server with the following, replacing the remote username and IP address with the relevant values:

```
> ssh-copy-id user@123.45.56.78
```

It should prompt you to enter the password for the remote server, after which point you should be able to use scp commands without a password.

In my case the scp command to send the csv file looks like the following:

```
> scp /home/pi/dev/rasBuoy/node/coord/out.csv forc196@wyvern.cs.newpaltz.edu:~/WWW/rasbuoy/
```

Appendix B

Node.js files

```

***** router.js *****

// This is the main router file.
// It will run at boot time, open the serial port, and listen for incoming data.
// Every 15 minutes it will take a reading, write the database and send data.

var temp = require('./read_temp.js');
var config = require('./read_config.js');
var db = require('./write_db.js');
var incoming = require('./write_incoming.js');
var converter = require('./convert_data.js');
var xbee = require('./send_data.js');
var adc = require('./measure_voltage.js');
var uuidV4 = require('uuid/v4');
var serial = require('serialport');
var xbee_api = require('xbee-api');
var SerialPort = require('serialport');

var configValues;
var serialport;
var dataArray = {};

var xbeeAPI = new xbee_api.XBeeAPI({
  api_mode: 1
});

// read config values
config.readConfig(function(values){
  configValues = values;
  // get the serial port value
  var xbeePort = configValues['XBEE_PORT'];
  // open serial port
  serialport = new serial(xbeePort, {
    baudrate: 9600,
    parser: xbeeAPI.rawParser()
  });
  // listen for incoming data
  xbeeAPI.on("frame_object", function (frame) {
    if (frame['data'] != null) {
      var stringArray = [];

```

```

// split the data frame into pieces
var data = frame['data'].toString();
var stringPart = data.split("|")[0];
var identPart = data.split("|")[1];
var identifier = identPart.split("-")[0];
var section = identPart.split("-")[1];

// check if we received the final piece
var done = false;
if (section.indexOf("%") > -1) {
    // remove end identifier and mark section as done
    section = section.split("%")[0];
    done = true;
}

// get the current string array if it exists
if (dataArray[identifier] != null) {
    stringArray = dataArray[identifier];
}
// add current piece to string array
stringArray[section] = stringPart;
dataArray[identifier] = stringArray;

// process the data when all pieces are received
if (done) {
    // rebuild the string
    var dataString = "";
    for (chunk in stringArray) {
        dataString += stringArray[chunk];
    }

    // get data values from string
    var values = dataString.split("<=>");

    // build dictionary from values
    var valueArray = {};
    var sensorNum = 0;
    for (var i = 0; i < values.length; i++) {
        if (values[i].indexOf("SENSOR_ID") > -1) {
            var sensorArray = {};
            for (var j = i; j < i + 6; j++) {
                var key = values[j].split("=")[0];
                var value = values[j].split("=")[1];
                sensorArray[key] = value;
            }

```

```

        i = i + 5;
        valueArray['SENSOR_'+sensorNum++] = sensorArray;
    } else {
        var key = values[i].split("=")[0];
        var value = values[i].split("=")[1];
        valueArray[key] = value;
    }
}
// enter values into the database
var duplicate = incoming.writeDb(valueArray);

// send data if it is new
if (!duplicate) {
    // convert data to byte string
    var byteData = converter.arrayToByteString(valueArray);
    // send data to xbee
    sendToXbee(byteData, serialport, 0)
}
}
});

// start the sensor reading cycle
sleep(60000).then(()=>{
    startReading();
});

// read the sensor every fifteen minutes
function startReading() {
    readSensor();
    sleep(900000).then(()=>{
        startReading();
    });
}

// takes a reading from the temperature sensor
function readSensor() {
    // read data from sensors
    var tempData = temp.readTemps();

    // add tempData to configData
    for (key in configValues) {
        if (key.indexOf("SENSOR_") > -1) {
            var sensorID = configValues[key]['SENSOR_ID'];
            for (id in tempData) {

```

```

    if (id == sensorID) {
        configValues[key]['MEASUREMENT_VALUE'] = tempData[id];
    }
}
// add measurement ID
configValues[key]['ID'] = uuidV4();
}
}

// get datetime
configValues['DATE'] = new Date().toISOString().slice(0, 19).replace('T', ' ');
// get xbee port
xbeePort = configValues['XBEE_PORT'];
// measure voltage
adc.readVoltage(function(voltage){
    configValues['VOLTAGE'] = voltage;
    // write to database
    db.writeDb(configValues);
    // convert data to byte string
    var byteData = converter.arrayToByteString(configValues);
    // send data to xbee
    sendToXbee(byteData,serialport,0)
});
}

// sends part of array to xbee, sleeps for 5 seconds to ensure that
// serial port is clear, and recurses with next part of data
function sendToXbee(byteData,serialport,i) {
    if (i < byteData.length) {
        xbee.sendData(byteData[i],serialport);
        sleep(5000).then(()=>{
            sendToXbee(byteData,serialport,++i);
        });
    }
}

function sleep(time) {
    return new Promise((resolve) => setTimeout(resolve, time));
}

```

```

***** write_incoming.js *****

// This module writes incoming sensor data to the database

var mysql = require('mysql');

exports.writeDb = function(configValues) {

  // get database values
  var host = configValues['HOST'];
  var user = configValues['USER'];
  var password = configValues['PASSWORD'];
  var database = configValues['DATABASE'];
  var table = configValues['TABLE'];

  var connection = mysql.createConnection({
    host    : host,
    user    : user,
    password : password,
    database : database
  });

  connection.connect();

  // get node data
  var nodeID = configValues['NODE_ID'];
  var latitude = configValues['LATITUDE'];
  var longitude = configValues['LONGITUDE'];
  var environment = configValues['ENVIRONMENT'];

  // get date
  var date = configValues['DATE'];

  // get voltage if present
  var voltage = configValues['VOLTAGE'];

  // get sensor data if it exists

  for (key in configValues) {
    if (configValues[key] != null) {
      if (configValues[key]['MEASUREMENT_VALUE'] != null) {
        var sensorArray = configValues[key];
        var sensorID = sensorArray['SENSOR_ID'];
        var depth = sensorArray['DEPTH_M'];
        var measurementType = sensorArray['MEASUREMENT_TYPE'];
        var measurementUnits = sensorArray['MEASUREMENT_UNITS'];

```



```

var measurementValue = sensorArray['MEASUREMENT_VALUE'];
var uuid = sensorArray['ID'];
var select = 'SELECT COUNT(*) as count FROM '+table+' WHERE id=?';

// check if uuid already database
getUuid(connection, select, uuid, function(results){
    var count = results[0].count;
    if (count == 0) {
        var dataSet =
{id:uuid,node_id:nodeID,datetime:date,latitude:latitude,longitude:longitude,voltage:voltage,environme
nt:environment,sensor_id:sensorID,depth_m:depth,measurement_type:measurementType,measurement
_units:measurementUnits,measurement_value:measurementValue};

        var query = connection.query('INSERT INTO '+table+' SET ?',
                                     dataSet, function (error, results) {
                    if (error) throw error;
                });
        //console.log(query.sql);
    } else {
        // return if data is already in database
        connection.end();
        return true;
    }
    connection.end();
    return false;
});
}
}
}

function getUuid(connection, select, uuid, callback) {
    // check if uuid is already in database
    connection.query(select, [uuid], function(error,results,fields) {
        callback(results);
    });
}

```

```
***** send_data.js *****

// This module will send byte data through the XBee

var serial = require('serialport');
var xbee = require('xbee-api');

var xbeeAPI = new xbee.XBeeAPI({
  api_mode: 1
});

exports.sendData = function(data, serialport) {
  var frame_obj = {
    type: 0x10,
    id: 0x01,
    broadcastRadius: 0x00,
    options: 0x00,
    data: chunk
  };

  serialport.write(xbeeAPI.buildFrame(frame_obj));
};
```

```
***** convert_data.js *****
```

```
// This module will convert a data array to a byte string
```

```
exports.arrayToByteString = function(data) {
  // read data from array
  // node id, lat, lng, env, voltage, date, sensor id, mtype, munits, mvalue, depth
```

```
  var dataString = "";
  for (key in data) {
    if (key.indexOf("SENSOR_") > -1) {
      if (data[key]['MEASUREMENT_VALUE'] != null) {
        for (sensorKey in data[key]) {
          dataString += sensorKey+"="+data[key][sensorKey]+"<=>";
        }
      }
    } else {
      dataString += key+"="+data[key]+"<=>";
    }
  }
}
```

```
// remove trailing separator and convert to byte string
dataString = dataString.substring(0, dataString.length - 3);
// split string, 80 characters per chunk
var byteData = [];
var k = 0;
var id = getIdentifier(0,1000000);
```

```
for (var i = 0; i < dataString.length; i=i) {
  var data = [];
  if (i + 80 > dataString.length) {
    j = dataString.length;
  } else {
    j = i + 80;
  }
}
```

```
var chunk = dataString.substring(i,j);
i += 80;
```

```
// add identifier to data chunk
chunk += "|" + id + "-" + k++;
// add end identifier
if (j == dataString.length) {
  chunk += "%";
}
```

```
    data.push(chunk);
    byteData.push(toByteArray(data));
  }
  return byteData;
}

// convert the data string to a byte array
function toByteArray(data) {
  var bytes = [];
  var i;
  for (i = 0; i < data[0].length; ++i) {
    bytes.push(data[0].charCodeAt(i));
  }
  return bytes;
}

// generate a random data identifier
function getIdentifier(min, max) {
  min = Math.ceil(min);
  max = Math.floor(max);
  return Math.floor(Math.random() * (max - min)) + min;
}
```

```
***** write_db.js *****
```

```
// This module writes the sensor data to the database
```

```
var mysql = require('mysql');
```

```
exports.writeDb = function(configValues) {
```

```
  // get database values
```

```
  var host = configValues['HOST'];
```

```
  var user = configValues['USER'];
```

```
  var password = configValues['PASSWORD'];
```

```
  var database = configValues['DATABASE'];
```

```
  var table = configValues['TABLE'];
```

```
  var connection = mysql.createConnection({
```

```
    host    : host,
```

```
    user    : user,
```

```
    password : password,
```

```
    database : database
```

```
  });
```

```
  connection.connect();
```

```
  // get node data
```

```
  var nodeID = configValues['NODE_ID'];
```

```
  var latitude = configValues['LATITUDE'];
```

```
  var longitude = configValues['LONGITUDE'];
```

```
  var environment = configValues['ENVIRONMENT'];
```

```
  // get date
```

```
  var date = configValues['DATE'];
```

```
  // get voltage if present
```

```
  var voltage = configValues['VOLTAGE'];
```

```
  // get sensor data if it exists
```

```
  for (key in configValues) {
```

```
    if (configValues[key]['MEASUREMENT_VALUE'] != null) {
```

```
      var sensorArray = configValues[key];
```

```
      var sensorID = sensorArray['SENSOR_ID'];
```

```
      var depth = sensorArray['DEPTH_M'];
```

```
      var measurementType = sensorArray['MEASUREMENT_TYPE'];
```

```
      var measurementUnits = sensorArray['MEASUREMENT_UNITS'];
```

```
      var measurementValue = sensorArray['MEASUREMENT_VALUE'];
```

```
      var uuid = sensorArray['ID'];
```

```
    var dataSet =
    {id:uuid,node_id:nodeID,datetime:date,latitude:latitude,longitude:longitude,voltage:voltage,environment:environment,sensor_id:sensorID,depth_m:depth,measurement_type:measurementType,measurement_units:measurementUnits,measurement_value:measurementValue};

    var query = connection.query('INSERT INTO '+table+' SET ?', dataSet, function (error, results) {
        if (error) throw error;
    });
    //console.log(query.sql);
}
}
connection.end();
}
```

```
***** read_temp.js *****
```

```
// This module returns an array of data from all attached temperature sensors
```

```
var sensor = require('ds18x20');
```

```
// reads the temperatures from the sensors
```

```
exports.readTemps = function () {
```

```
  tempData = sensor.getAll();
```

```
  return tempData;
```

```
};
```

```
***** read_config.js *****
```

```
// This module will read values from router.conf
```

```
var lineReader = require('line-reader');
```

```
exports.readConfig = function(callback) {
```

```
    var configValues = {};
```

```
    var sensorId = 0;
```

```
    var lines = [];
```

```
    // read lines config into array
```

```
    lineReader.eachLine('/home/pi/dev/rasBuoy/node/router/router.conf', function(line, last) {
```

```
        lines.push(line);
```

```
        if (last) {
```

```
            // get data from config file
```

```
            for (var i = 0; i < lines.length; i++) {
```

```
                var line = lines[i];
```

```
                // xbee port
```

```
                if (line.indexOf("XBEE_PORT") > -1) {
```

```
                    var xbee_port = line.split("=")[1];
```

```
                    if (xbee_port.length > 0) {
```

```
                        configValues["XBEE_PORT"] = xbee_port;
```

```
                    }
```

```
                }
```

```
            // database values
```

```
            if (line.indexOf("HOST") > -1) {
```

```
                var host = line.split("=")[1];
```

```
                if (host.length > 0) {
```

```
                    configValues["HOST"] = host;
```

```
                }
```

```
            }
```

```
            if (line.indexOf("USER") > -1) {
```

```
                var user = line.split("=")[1];
```

```
                if (user.length > 0) {
```

```
                    configValues["USER"] = user;
```

```
                }
```

```
            }
```

```
            if (line.indexOf("PASSWORD") > -1) {
```

```
                var password = line.split("=")[1];
```

```
                if (password.length > 0) {
```

```
                    configValues["PASSWORD"] = password;
```

```
                }
```

```
            }
```



```

if (line.indexOf("DATABASE") > -1) {
    var database = line.split("=")[1];
    if (database.length > 0) {
        configValues["DATABASE"] = database;
    }
}
if (line.indexOf("TABLE") > -1) {
    var table = line.split("=")[1];
    if (table.length > 0) {
        configValues["TABLE"] = table;
    }
}

// node values
if (line.indexOf("NODE_ID") > -1) {
    var nodeID = line.split("=")[1];
    if (nodeID.length > 0) {
        configValues["NODE_ID"] = nodeID;
    }
}
if (line.indexOf("LATITUDE") > -1) {
    var latitude = line.split("=")[1];
    if (latitude.length > 0) {
        configValues["LATITUDE"] = latitude;
    }
}
if (line.indexOf("LONGITUDE") > -1) {
    var longitude = line.split("=")[1];
    if (longitude.length > 0) {
        configValues["LONGITUDE"] = longitude;
    }
}
if (line.indexOf("ENVIRONMENT") > -1) {
    var environment = line.split("=")[1];
    if (environment.length > 0) {
        configValues["ENVIRONMENT"] = environment;
    }
}

// sensor values
if (line.indexOf("SENSOR_ID") > -1) {
    var sensorValues = {};
    var sensorID = line.split("=")[1];
    if (sensorID.length > 0) {
        sensorValues["SENSOR_ID"] = sensorID;
    }
}

```

```

    line = lines[++i];
    var depth = line.split("=")[1];
    if (depth.length > 0) {
        sensorValues["DEPTH_M"] = depth;
    }
    line = lines[++i];
    var measurementType = line.split("=")[1];
    if (measurementType.length > 0) {
        sensorValues["MEASUREMENT_TYPE"] = measurementType;
    }
    line = lines[++i];
    var measurementUnits = line.split("=")[1];
    if (measurementUnits.length > 0) {
        sensorValues["MEASUREMENT_UNITS"] = measurementUnits;
    }
    configValues["SENSOR_"+sensorId++] = sensorValues;
}
}
callback(configValues);
}
});
};

```

```
***** router.conf *****
```

```
# This file contains information about the node and it's sensors
```

```
# XBee port value (/dev/serialx, /dev/ttyAMAx, /dev/ttyUSBx)
```

```
XBEE_PORT=
```

```
# Database values
```

```
HOST=
```

```
USER=
```

```
PASSWORD=
```

```
DATABASE=
```

```
TABLE=
```

```
# Node values
```

```
# Node ID: This is a unique integer identifier for each
```

```
# sensor node. Nodes can simply be numbered 0,1,2,etc.
```

```
# Lat/Long: If you know the lat/long of your node, enter
```

```
# it here. If you have a GPS running on your node
```

```
# the values will be recorded here.
```

```
# Environment: This is a description of where your node
```

```
# is located (i.e., lake edge, forest, office, etc).
```

```
NODE_ID=
```

```
LATITUDE=
```

```
LONGITUDE=
```

```
ENVIRONMENT=
```

```
# Sensor values
```

```
# Sensor ID: This is a unique identifier that will be entered
```

```
# into the database along with the sensor data. If your
```

```
# sensor has a device ID associated with it, use that.
```

```
# Depth: This is the depth, in meters, of your sensor if it
```

```
# is submerged in water. If it is not submerged, enter NA.
```

```
# Measurement Type: This a description of what you are measuring,
```

```
# (i.e., air temperature, water temperature, etc).
```

```
# Measurement Units: The units of measurement.
```

```
SENSOR_ID=
```

```
DEPTH_M=
```

```
MEASUREMENT_TYPE=
```

```
MEASUREMENT_UNITS=
```

```
***** measure_voltage.js *****
```

```
var ads1x15 = require('node-ads1x15');
```

```
var chip = 1; //0 for ads1015, 1 for ads1115  
var adc = new ads1x15(chip);
```

```
var chA = 0; //channel 0, 1, 2, or 3...  
var chB = 1;  
var samplesPerSecond = '250';  
var progGainAmp = '4096';
```

```
var reading = 0;
```

```
exports.readVoltage = function(callback) {  
  if(!adc.busy)  
  {  
    adc.readADCDifferential(chA, chB, progGainAmp, samplesPerSecond, function(err, data) {  
      if(err)  
      {  
        throw err;  
      }  
      reading = Math.round(data)/1000;  
      callback(reading);  
    });  
  }  
};
```

```
***** gps_check.js *****
```

```
// This module will query the gps and write latitude and longitude to router.conf
```

```
var gpsd = require('node-gpsd');
var lineReader = require('line-reader');
var fs = require('fs');
```

```
var latitudes = [];
var longitudes = [];
var daemons = [];
```

```
// create a gps daemon on the specified port
```

```
function createDaemon(port) {
  var device = '/dev/ttyUSB'+port;
  // start the gps daemon
  var daemon = new gpsd.Daemon({
    program: 'gpsd',
    device: device,
    port: 2947,
    pid: '/tmp/gpsd.pid',
    readOnly: false,
    logger: {
      info: function() {},
      warn: console.warn,
      error: console.error
    }
  });
  // add daemon to array
  daemons.push(daemon);
}
```

```
// start a daemon on the specified port
```

```
function startDaemon(port,daemon) {
  daemon.start(function() {
    console.log('Started Daemon on USB'+port);
  });
}
```

```
// attempt to kill a daemon on the specified port
```

```
function stopDaemon(port,daemon) {
  try {
    daemon.stop(function() {
      console.log('Killed Daemon on USB'+port);
    });
  } catch (err) {
```

```

        console.log("Daemon already killed");
    }
}

// create a daemon on USB ports 0-3
// gps will probably be on USB0, but we will check them all
for (var i = 0; i < 4; i++) {
    createDaemon(i);
}

// start a daemon on USB ports 0-3
for (var i = 0; i < 4; i++) {
    startDaemon(i, daemons[i]);
}

// create a gpsd listener
var listener = new gpsd.Listener({
    port: 2947,
    hostname: 'localhost',
    logger: {
        info: function() {},
        warn: console.warn,
        error: console.error
    },
    parse: true
});

// connect listener
listener.connect(function() {
    console.log('Listener Connected');
});

// TPV data emitted if parse is 'true'
listener.on("TPV", function(data) {
    //console.log(data);
    // add lat/long to array
    if (data['lat'] != undefined) {
        latitudes.push(data['lat']);
        longitudes.push(data['lon']);
    }
    // check array size, stop at 10
    if (latitudes.length == 10) {
        var latitude = 0;
        var longitude = 0;
        // average values, output to file, exit
        for (var i in latitudes) {

```

```

        latitude += latitudes[i];
        longitude += longitudes[i];
    }
    latitude /= 10;
    longitude /= 10;
    // round to 8 decimal places
    latitude = latitude.toFixed(8);
    longitude = longitude.toFixed(8);
    // write output to router.conf
    writeLocation(latitude,longitude);
    // kill all the daemons
    for (var i = 0; i < 4; i++) {
        stopDaemon(i,daemons[i]);
    }
}
});

// raw data emitted if parse is 'false'
//listener.on('raw', function(data) {
// console.log(data);
//});

listener.watch({class: 'WATCH', json: true, nmea: false});

// set a timeout function for 30 seconds
setTimeout(function(){
    console.log('GPS timeout');
    // kill all the daemons
    for (var i = 0; i < 4; i++) {
        stopDaemon(i,daemons[i]);
    }
    process.exit();
},30000);

// writes the location to router.conf
writeLocation(latitude,longitude) {
    lines = [];
    // read router.conf into array
    lineReader.eachLine('/home/pi/dev/rasBuoy/node/router/router.conf', function(line, last) {
        lines.push(line);
        if (last) {
            var writeData = "";
            // find and modify lat/long
            for (var i = 0; i < lines.length; i++) {
                if (lines[i].indexOf("LATITUDE") > -1) {
                    lines[i] = "LATITUDE="+latitude;

```

```
    }
    if (lines[i].indexOf("LONGITUDE") > -1) {
        lines[i] = "LONGITUDE="+longitude;
    }
    // append line to write data
    writeData += lines[i]+"\\n"
}
// write data to router.conf
fs.writeFile('/home/pi/dev/rasBuoy/node/router/router.conf', writeData, function(err){
    if(err){ console.log(err);}
});
return false; // stop reading
}
});
};
```



```

***** coordinator.js *****

// This is the main coordinator module.
// It runs at boot time and listens for incoming data.

var SerialPort = require('serialport');
var xbee_api = require('xbee-api');
var db = require('./write_incoming.js');
var config = require('./read_config.js');

var dataArray = {};

var xbee = new xbee_api.XBeeAPI({
  api_mode: 1
});

// read config values
config.readConfig(function(values){
  configValues = values;

  // get serial port
  var xbeePort = configValues['XBEE_PORT'];

  // get the xbee serial port
  var serialport = new SerialPort(xbeePort, {
    baudrate: 9600,
    parser: xbee.rawParser()
  });

  // look for incoming data on serial port
  xbee.on("frame_object", function (frame) {
    var stringArray = [];

    // split the data frame into pieces
    var data = frame['data'].toString();
    var stringPart = data.split("|")[0];
    var identPart = data.split("|")[1];
    var identifier = identPart.split("-")[0];
    var section = identPart.split("-")[1];

    // check if we received the final piece
    var done = false;
    if (section.indexOf("%") > -1) {
      // remove end identifier and mark section as done
      section = section.split("%")[0];
      done = true;
    }
  });
}

```

```

    }

    // get the current string array if it exists
    if (dataArray[identifier] != null) {
        stringArray = dataArray[identifier];
    }
    // add current piece to string array
    stringArray[section] = stringPart;
    dataArray[identifier] = stringArray;

    // process the data when all pieces are received
    if (done) {
        // rebuild the string
        var dataString = "";
        for (chunk in stringArray) {
            dataString += stringArray[chunk];
        }

        // get data values from string
        var values = dataString.split("<=>");

        // build dictionary from values
        var valueArray = { };
        var sensorNum = 0;
        for (var i = 0; i < values.length; i++) {
            if (values[i].indexOf("SENSOR_ID") > -1) {
                var sensorArray = { };
                for (var j = i; j < i + 6; j++) {
                    var key = values[j].split("=")[0];
                    var value = values[j].split("=")[1];
                    sensorArray[key] = value;
                }
                i = i + 5;
                valueArray['SENSOR_'+sensorNum++] = sensorArray;
            } else {
                var key = values[i].split("=")[0];
                var value = values[i].split("=")[1];
                valueArray[key] = value;
            }
        }
        // enter values into the database
        db.writeDb(valueArray);
    }
});
});

```

```

***** write_incoming.js *****

// This module writes incoming sensor data to the database

var mysql = require('mysql');

exports.writeDb = function(configValues) {

  // get database values
  var host = configValues['HOST'];
  var user = configValues['USER'];
  var password = configValues['PASSWORD'];
  var database = configValues['DATABASE'];
  var table = configValues['TABLE'];

  var connection = mysql.createConnection({
    host    : host,
    user    : user,
    password : password,
    database : database
  });

  connection.connect();

  // get node data
  var nodeID = configValues['NODE_ID'];
  var latitude = configValues['LATITUDE'];
  var longitude = configValues['LONGITUDE'];
  var environment = configValues['ENVIRONMENT'];

  // get date
  var date = configValues['DATE'];

  // get voltage if present
  var voltage = configValues['VOLTAGE'];

  // get sensor data if it exists

  for (key in configValues) {
    if (configValues[key] != null) {
      if (configValues[key]['MEASUREMENT_VALUE'] != null) {
        var sensorArray = configValues[key];
        var sensorID = sensorArray['SENSOR_ID'];
        var depth = sensorArray['DEPTH_M'];
        var measurementType = sensorArray['MEASUREMENT_TYPE'];
        var measurementUnits = sensorArray['MEASUREMENT_UNITS'];

```

```

var measurementValue = sensorArray['MEASUREMENT_VALUE'];
var uuid = sensorArray['ID'];
var select = 'SELECT COUNT(*) as count FROM '+table+' WHERE id=?';

// check if uuid already database
getUuid(connection, select, uuid, function(results){
    var count = results[0].count;
    if (count == 0) {
        var dataSet =
{id:uuid,node_id:nodeID,datetime:date,latitude:latitude,longitude:longitude,voltage:voltage,environme
nt:environment,sensor_id:sensorID,depth_m:depth,measurement_type:measurementType,measurement
_units:measurementUnits,measurement_value:measurementValue};

        var query = connection.query('INSERT INTO '+table+' SET ?',
dataSet, function (error, results) {
            if (error) throw error;
        });
        //console.log(query.sql);
    }
    connection.end();
});
    }
}
}

function getUuid(connection, select, uuid, callback) {
    // check if uuid is already in database
    connection.query(select, [uuid], function(error,results,fields) {
        callback(results);
    });
}

```

```
***** db_to_csv.js *****
```

```
// This module will read from the database and output a csv file
```

```
var mysql = require('mysql');
var config = require('./read_config.js');
var fs = require('fs');
```

```
function exportDb() {
  // get database settings from config file
  config.readConfig(function(values){
    var configValues = values;

    // get database values
    var host = configValues['HOST'];
    var user = configValues['USER'];
    var password = configValues['PASSWORD'];
    var database = configValues['DATABASE'];
    var table = configValues['TABLE'];

    var connection = mysql.createConnection({
      host    : host,
      user    : user,
      password : password,
      database : database
    });

    connection.connect();

    // query the database values
    var query = 'SELECT node_id,datetime,sensor_id,measurement_value FROM '+table+' ORDER
    BY datetime';

    connection.query(query, function(error,results,fields) {
      var resultsArray = [];
      for (index in results) {
        var resultArray = [];
        for (key in results[index]) {
          resultArray.push(results[index][key]);
        }
        resultsArray.push(resultArray);
      }
      // write database values to csv files
      // out0.csv and out1.csv contain datetime and temperature values from nodes 0 and 1
      // outvoltage0.csv contains datetime and voltage values from node 0
      // latest_temps.csv contains the latest temperature reading from nodes 0 and 1
    });
  });
}
```

```

var outData0 = "";
var outData1 = "";
var outVolt0 = "";
var lastData0 = [];
var lastData1 = [];
for (index in resultsArray) {
  var nodeId = resultsArray[index][0];
  var entries = resultsArray[index];
  if (nodeId == 0) {
    lastData0 = entries;
  } else {
    lastData1 = entries;
  }
  for (var i = 1; i < entries.length; i++) {
    var entry = entries[i];
    if (i == 0 || i == 1) { // node id and date
      if (nodeId == 0) {
        outData0 += entry+", ";
        outVolt0 += entry+", ";
      } else {
        outData1 += entry+", ";
      }
    }
    if (i == 2) { // temperature
      if (nodeId == 0) {
        outData0 += entry+", ";
      } else {
        outData1 += entry+", ";
      }
    }
    if (i == 3 && nodeId == 0) { // voltage
      outVolt0 += entry+", ";
    }
  }
  if (nodeId == 0) {
    outData0 = outData0.substring(0, outData0.length-1)+'\n';
    outVolt0 = outVolt0.substring(0, outVolt0.length-1)+'\n';
  } else {
    outData1 = outData1.substring(0, outData1.length-1)+'\n';
  }
}
var lastData = lastData0+"\n"+lastData1;
fs.writeFile('/home/pi/dev/rasBuoy/node/coord/out0.csv', outData0, 'utf8', function(err) {
  if (err) {
    console.log(err);
  }
}

```

```
});  
fs.writeFile('/home/pi/dev/rasBuoy/node/coord/outvolt0.csv', outVolt0, 'utf8', function(err) {  
  if (err) {  
    console.log(err);  
  }  
});  
fs.writeFile('/home/pi/dev/rasBuoy/node/coord/out1.csv', outData1, 'utf8', function(err) {  
  if (err) {  
    console.log(err);  
  }  
});  
fs.writeFile('/home/pi/dev/rasBuoy/node/coord/latest_temps.csv', lastData, 'utf8', function(err) {  
  if (err) {  
    console.log(err);  
  }  
});  
connection.end();  
});  
});  
}  
  
exportDb();
```

```
***** send_data.sh *****
```

```
#!/bin/bash
```

```
scp /home/pi/dev/rasBuoy/node/coord/out0.csv forcel96@wyvern.cs.newpaltz.edu:~/WWW/rasbuoy/  
scp /home/pi/dev/rasBuoy/node/coord/out1.csv forcel96@wyvern.cs.newpaltz.edu:~/WWW/rasbuoy/  
scp /home/pi/dev/rasBuoy/node/coord/latest_temps.csv  
forcel96@wyvern.cs.newpaltz.edu:~/WWW/rasbuoy/  
scp /home/pi/dev/rasBuoy/node/coord/outvolt0.csv  
forcel96@wyvern.cs.newpaltz.edu:~/WWW/rasbuoy/
```


BIBLIOGRAPHY

- [1] John Cook, Naomi Oreskes, Peter T Doran, William R L Anderegg, Bart Verheggen, Ed W Maibach, J Stuart Carlton, Stephan Lewandowsky, Andrew G Skuce, Sarah A Green. Consensus on consensus: a synthesis of consensus estimates on human-caused global warming.
- [2] K. A. Garrett, G. A. Forbes, S. Savary, P. Skelsey, A. H. Sparks, C. Valdivia, A. H. C. van Bruggen, L. Willocquet, A. Djurle, E. Duveiller, H. Eckersten, S. Pande, C. Vera Cruz, J. Yuen. Complexity in climate-change impacts: an analytical framework for effects mediated by plant disease.
- [3] Brian R. Pickard, Jeremy Baynes, Megan Mehaffey, Anne C. Neale, "Translating Big Data into Big Climate Ideas," *The Solutions Journal*, Volume 6, Issue 1, January 2015.
- [4] John H. Porter Email the author John H. Porter, Paul C. Hanson, Chau-Chin Lin. "Staying afloat in the sensor data deluge." *Trends in Ecology and Evolution*. Volume 27, Issue 2, p121–129, February 2012.
- [5] Pille Meinson, Agron Idrizaj, Peeter Nõges, Tiina Nõges, Alo Laas. Continuous and high-frequency measurements in limnology: history, applications, and future challenges.
- [6] L. A. Jones, J. S. Kimball, K. C. McDonald, S. T. K. Chan, E. G. Njoku and W. C. Oechel, "Satellite Microwave Remote Sensing of Boreal and Arctic Soil Temperatures From AMSR-E," in *IEEE Transactions on Geoscience and Remote Sensing*, vol. 45, no. 7, pp. 2004-2018, July 2007.
- [7] M. Dunbabin and L. Marques, "Robots for Environmental Monitoring: Significant Advancements and Applications," in *IEEE Robotics & Automation Magazine*, vol. 19, no. 1, pp. 24-39, March 2012.
- [8] GLEON Membership Statistics. <http://gleon.org/members/GLEON-membership-statistics>. April 15, 2017.
- [9] Cook, B. I., Cook, E. R., Huth, P. C., Thompson, J. E., Forster, A., & Smiley, D. (2008). "A cross- taxa phenological dataset from Mohonk Lake, NY and its relationship to climate." *International Journal of Climatology*, 28(10), 1369-1383.
- [10] Mohonk Lake | GLEON. <http://gleon.org/node/4467>. April 15, 2017.
- [11] Rick Bonney, Caren B. Cooper, Janis Dickinson, Steve Kelling, Tina Phillips, Kenneth V. Rosenberg, and Jennifer Shirk. "Citizen Science: A Developing Tool for Expanding Science Knowledge and Scientific Literacy." *BioScience* 2009 59 (11), 977-984.
- [12] "About the Great Backyard Bird Count." Audubon, 13 Mar. 2017. <http://www.audubon.org/content/about-great-backyard-bird-count>. April 15, 2017.
- [13] Matthew C. Van de Bogert, Darren L. Bade, Stephen R. Carpenter, Jonathan J. Cole, Michael L. Pace, Paul C. Hanson, and Owen C. Langman. Spatial heterogeneity strongly affects estimates of ecosystem metabolism in two north temperate lakes.

- [14] Stacey Kuznetsov, Eric Paulos. "Rise of the expert amateur: DIY projects, communities, and cultures." NordiCHI 2010 Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries, Pages 295-304.
- [15] Miguel Gandra, Rui Seabra, Fernando P. Lima. A low-cost, versatile data logging system for ecological applications.
- [16] Faludi, Robert. Building wireless sensor networks: with ZigBee, XBee, Arduino, and processing. Farnham: O'Reilly Media, 2010.
- [17] David M. Charifson, Paul C. Huth, John E. Thompson, Robert K. Angyal, Michael J. Flaherty, David C. Richardson. "History of Fish Presence and Absence Following Lake Acidification and Recovery in Lake Minnewaska, Shawangunk Ridge, NY." Northeastern Naturalist, Volume 22, Issue 4 (2015): 762–781