WASP
Wireless Ad-hoc Sensor Protocol

Group SW513E15

Christian Lundtofte Sørensen
Henrik Djernes Thomsen
Jonathan Hastrup

Bjørn Opstad
Morten Mandrup Hansen
Mathias Corlin Mikkelsen

AALBORG UNIVERSITY
STUDENT REPORT

Department of computer science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø

Title:
WASP - Wireless Ad-hoc Sensor Protocol

Theme:
Embedded Systems

Project period:
02/08/2015 -
21/12/2015

Project group:
SW513E15

Members:
Christian Lundtofte Sørensen
Henrik Djernes Thomsen
Jonathan Hastrup
Bjørn Opstad
Morten Mandrup Hansen
Mathias Corlin Mikkelsen

Supervisor:
Hua Lu

No. of printed Copies: 8
No. of Pages: 82
No. of Appendix Pages: 12
Total no. of pages: 94
Completed: 18/12/2015

Synopsis:

The goal for this project is to design, model and implement a protocol for an embedded system, transmitting sensor data to a main node. An interview with a greenkeeper at Aalborg Golfcourse led to the purpose of the implemented protocol and devices used; monitoring water moisture levels in the soil.
An analysis of this problem domain was performed, and this laid the foundation for the design of the protocol and the technology to implement it. This report describes the process of analyzing, designing, implemenenting, and testing the protocol running on embedded systems.
The protocol is implemented in C++ on the devices and using radio modules for transfering data between nodes. Exponential backoff is used to make sure packets will arrive and to avoid deadlocks in the system.

Bjørn Opstad

Christian Lundtofte Sørensen

Morten Mandrup Hansen

Henrik Djernes Thomsen

Matihas Corlin Mikkelsen

Jonathan Hastrup

1. Reading instructions

This chapter contains definitions for some of the terms throughout the report. The report is intended to be read in sequential order, to fully comprehend the content and development.

Definition list

Through out the report several terms will appear frequently. These terms will be presented and described here to avoid confusion.

Sensor node
: The nodes in the network used to measure sensor data.

Main node
: The primary node that handles data from sensor nodes.

Radio module
: This refers to the wireless communication component used in the solution, which is tasked with sending and receiving data from and to other radio modules.

Packet
: A message, consisting of data and metadata, transmitted using radio modules between two nodes.

Contents

2. Introduction

Communication technology can today be seen as a foundation for a wide range of technologies. Devices today can communicate with each other by many means, and can take part in large and small networks alike. This report revolves around embedded systems and how communication can be established between these.

The embedded system context will be related to golf courses. As golf courses covers large areas, and the ground throughout the area contains information relevant for the golf course's maintenance personnel, an embedded system implementation could potentially save both time and resources.

To cover a large area, multiple devices would be required to measure at multiple locations. Communication will be utilized by having the devices send their data to a central device, avoiding users to manually extract data from each device.

This forms the initiating problem domain, and based on this the first part of the report will cover the analysis of this domain to determine and define a problem.

Initializing problem statement

To initialize and give foundation for an analysis, a problem statement is formulated:

Can embedded systems be utilized to assist in maintaining a golf course?

3. The working process

Since this project can be considered as the development of a professional software system, intended for use by someone other than the developer, rather than personal software development and usage, it can be considered software engineering[1]. As the developers in this project is a group of students, and a professional software system is usually developed in teams, this project could get support from various software engineering techniques.

In software engineering two types of software planning are usually relevant: Plan-driven and agile[1]. In the plan-driven process all activities are planned in advance, and progress is measured against this plan[1]. The agile process is incremental, and it is easier to return to previous activities and make changes

The flow chart in figure 3.1 shows this project's process model and displays the different stages of the project. This process model is a mix between the plan-driven and the agile method, but they are separated to different areas of the developing process. The project initializes in a plan-driven manner, while the implementation part is done in an agile manner.

The first two stages of the project is plan-driven. First the problem domain and application domain will be researched and analyzed. This enables the construction of requirements for the solution. The requirement are assumed not to change during the project period. In the next stage the requirements will be given priorities in order to determine the their importance for the final solution. The priorities are given by performing a MoSCoW analysis.

The next four stages are performed in an agile manner. Whenever the last of the four stages have been completed, which is the test stage, the time left until deadline is evaluated. If there is enough time left, then the four agile stages will be repeated. This is where the priorities that was given during the MoSCoW analysis becomes relevant. Each iteration will focus on a single tier of the requirements e.g. first iteration will focus on the "Must have" requirements. These will be the requirements in focus. If these requirements are considered met at the end of the four stages, then the next iteration will focus on the next tier of requirements, which in this example would be "Should have" priorities.

Research on how to meet the requirements in focus is done during the first stage of the agile stage followed by the second stage where the design of the solution will be developed. The designed solution will be implemented in the third stage.

The solution will be tested as a follow up to the implementation. Here it will be determined whether the requirements in focus are met. If the requirements were not met, then the requirements in focus will remain during the next iteration. If the requirements are met, then the next iteration will focus on the next tier of requirements. Another iteration will only be considered if there is sufficient time left until the deadline.

An agile implementation process is chosen, because the domain of the project is new and unknown for all group members, and therefore hard to plan consid-

ering time. Furthermore, this project has an impassable deadline, and therefore it would be hard for the group to set up a perfect set of requirements that would correspond to the project timespan. Keeping focus on a single tier at the time ensures that the most important features of the requirements are met first, which can then be expanded upon.

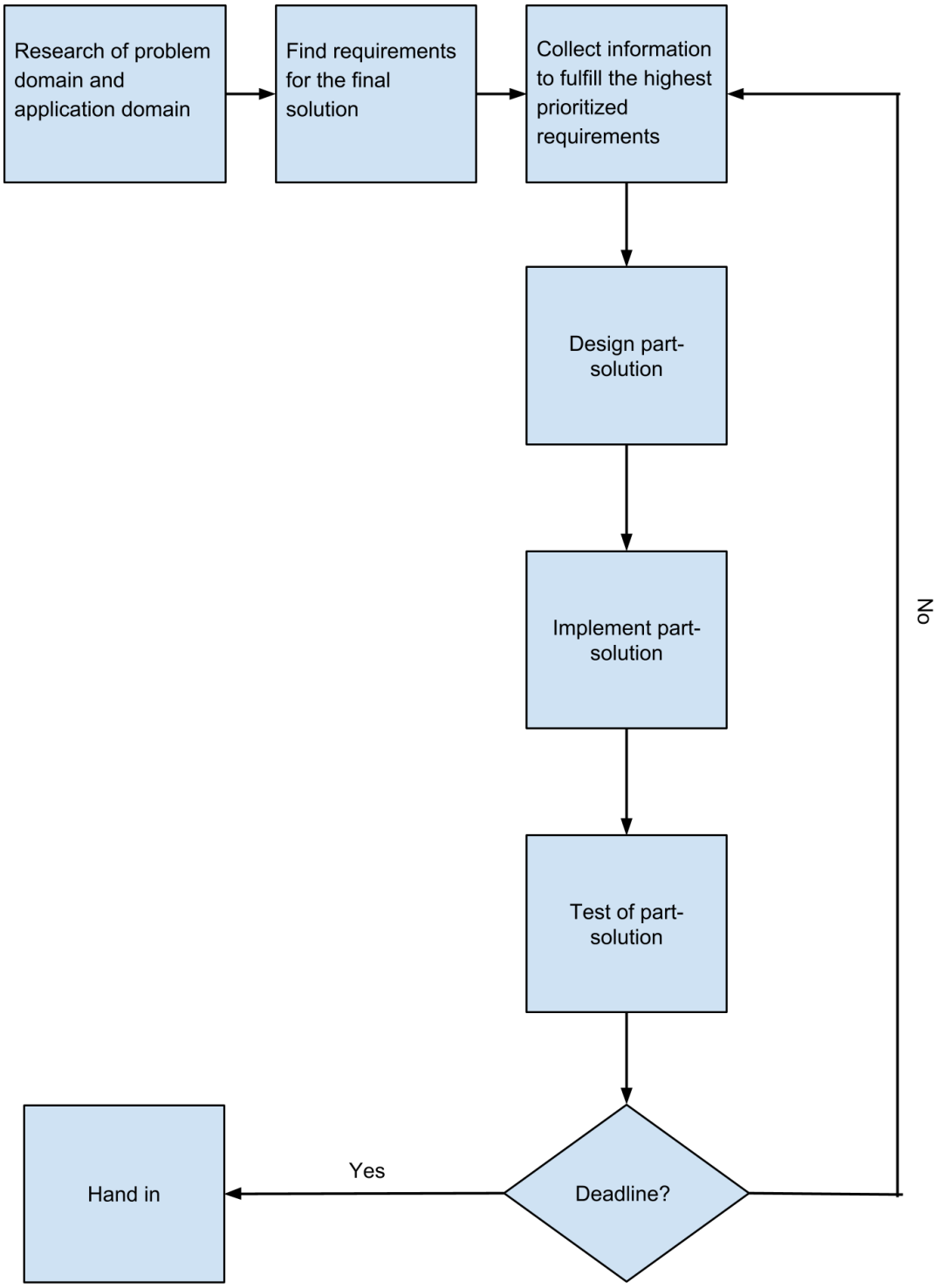

Figure 3.1: Process model diagram.

4. Analysis

To solve the broad initializing problem, specifying the problem is necessary. As the project group has little to no experience of golf course maintenance, an expert interview is chosen to enlighten the subject. This will be the foundation of the analysis. Thereafter, the problem domain is analyzed, covering the physical aspects of a golf course, existing technologies, security and power consumption.

The analysis results in a final problem statement and a requirement specification. These will give sufficient grounds for the design and implementation of the solution.

4.1 Interview

Two interviews were performed with a greenkeeper at Aalborg Golfcourse, Kim Jensen. The first being an initiating interview, and the second being a more in-depth interview. The interviewee provided insights on ideas for the system, requirements, potential problems, and additions for later iterations of the system. Everything presented in this interview is either information from the interviewee or conclusions of the information. The exact interview questions and answers can be found in appendix A.

In the early interview, the interviewee provided the initial idea of automating the gathering of data about the golf course. He suggested soil moisture, pH value in the soil, and pressure. These properties are currently monitored manually, and could be automated to save time for the greenkeepers.

Making the system easily scalable would be useful in case of changes in environment, in which case devices could be added to a new area, and the water levels could be measured more specifically.

The interviewee mentioned that a wireless system would be preferred over a wired system. At Aalborg Golfklub, the ground of the golf course already contains wires for both the lights and the watering system. A wireless solution would be simple to install compared to a wired system. Furthermore, no risk of cutting an important wire for the system would exist. A wireless system would be simpler to scale or restructure than a wired system.

Another consideration is where to place the devices. The sensors needs access to the soil in order to perform measurements, but they should not be placed above ground, as golfers could hit and damage the devices. This makes it necessary to bury the devices at known locations in the course. This is an important consideration because buried electronics in a humid environment is a challenge. Furthermore, if the system should be wireless as preferred, considerations about antenna quality should be made.

The depth of which the sensors are buried at is important, due to the different types of grass. The greens use a layer of sand at the top, which allows water to

quickly pass through. This makes it necessary to place the sensor at around 10-15cm. beneath the surface. To avoid wiring between the node and the sensor, the entire device should be buried at that depth.

In the interview, the interviewee mentioned several other possibilities for the project to save even more work. The pH-meter would allow the greenkeepers to create specific mixes of fertilizer for different parts of the course, depending on the pH value of the soil. The pressure sensor could help determining where on the course the soil has to be pricked, in order to keep the grass growing and water flowing.

4.2 Golf course

In this section, the physical and technical aspects of a golf course are described. These factors are analyzed to be able to define requirements for the solution and to formulate a problem statement.

Firstly, the physical and geographical aspect will be examined. Thereafter, special elements of the task are studied to determine hardware for the solution.

4.2.1 Environment

This subsection contains analysis and descriptions of golf courses' physical properties. Golf courses, as a problem domain, is considered static. This means that alterations to the original problem should not suddenly appear.

Physical characteristics
Golf courses have no strict geographical requirements, and each golf course is different, hence these characteristics are guidelines and conventions only, and not necessarily applicable to all golf courses. This section is mainly based on[2].

A golf course is typically covering a large area, and usually has 18 holes. As golf courses mainly are located outside, they are susceptible to weather and this requires possible devices placed on the golf course to be weather sealed, to avoid replacing them for example after a storm or heavy rain.

The area of the golf course contains different types of flora. There can be open fields of grass, wood areas, grass or rock hills of varying sizes, sand traps as well as ponds or lakes. Some of these areas can create difficulties for communication between devices, either by affecting wireless signals by delay, dampening or reverberation, or by being obstacles for laying cables.

Game areas
The different areas of the golf course are related to the different parts of the game. The *tee* is the start area. It usually has a level stance and short grass.

The *fairway* is the physically largest area and has longer grass. The *rough* is the part surrounding the main playing area and can also be played on, but does not require the same amount of maintenance as the other areas, in accordance with the name. The *green* is the destination area of a hole, and is closely mown and cared for.

The areas are treated differently and have varied need of attention because of the properties of grass and soil. The green requires detailed maintenance due to the precision necessary for the golfer's last strokes.

Other characteristics
Golf courses has people roaming, and there is a chance for objects on it getting hit by golf balls, as stated in the interview. Therefore, the devices to be placed on a golf course should be tough, replaceable, repairable or be placed in such a way that they avoid damage.

4.2.2 Grass and soil properties

A part of maintaining the grass is to consider the soils moisture [3]. This can be measured by several methods, for example by visible indications or direct assessment by a sensor. Currently, in accordance to the interviewee, this is monitored by using visual aids and indications and not by sensors.

Soil mostly consists of dirt and sand[4], and has properties like density, moisture, acidity and water throughput. The grass should be watered not too little and not too much, and the level of watering and other care depends on the properties of the soil. The properties can vary due to weather and season, and accordingly require more or less attention in care and treatment.

4.3 Existing technologies

This section contains an analysis of existing technologies used for golf course maintenance. The covered aspects are according to the interview answers. These technologies are analyzed to gain insight and inspiration.

4.3.1 Rain Bird SMRT-Y

Rain Bird produces the sensor SMRT-Y that measures soil moisture and displays it on a simple user interface[5].

The system from Rain Bird requires the use of Rain Bird irrigation, as the system is directly connected to the irrigation system. According to Rain Bird, this system provides over 40% in water savings[6].

SMRT-Y is fully automatic and requires no maintenance. Once buried it decides if the irrigation system should be turned on, based on the soil moisture. The only setup required is to set the threshold of the area where the sensor and irrigation system is located[6].

The sensors are small and therefore well suited to bury around the golf course. It is required to be in close proximity of irrigation valves, which might not be useful everywhere. The system does not send data to a central unit, making it difficult to determine the soil moisture, but instead makes decisions for the greenkeepers.

4.3.2 Toro Turf Guard™

The company Toro created the Turf Guard™ sensors that allows greenkeepers to keep track of sensor data in real time. The sensors measure moisture, temperature, and salinity in the soil[7]. The sensors are wireless and they have a range of 152 meters, but can be connected to repeaters to get a range of 1500 meters[7].

The sensors are designed with two layers of sensors, allowing them to measure the top soil and further down with one device.

The size of the device allows it to fit into a standard hole-size, making them well suited to install on greens where a tool for creating holes are used[8].

The entire system can be monitored from a computer, where the data are transmitted to and shown in a user interface with locations for sensors available[8].

The system requires the usage of other Toro systems to be working, making it difficult to implement if systems from another manufacturer is already used. It also requires the use of repeaters close to the sensors, for transmitting data to the main device[8].

Figure 4.1: Toro Turf Guard™ sensor[8].

4.4 Other issues

This section contains information about other potential issues that have not been directly encountered through the analysis, but still could be relevant for the final system.

4.4.1 Power consumption and range consideration

To be able to transmit data wirelessly, from all areas of a golf course to a node, the wireless range of the devices is considered.

Modules with a long range can require more power than a short range module, which would require more of a power supply. Furthermore, the modules could physically become larger, because of a bigger antenna, which would increase the amount of work in burying the devices in the ground. Because of the risk of being hit, and destroyed, by either a golf ball, or a lawn mower, it is not an option to place the entire device above ground. Using big antennas would make it easier to establish a direct wireless connection between nodes.

Another approach with short range modules can be considered. Short range radio modules usually requires less power than a long range module, thereby lowering the requirements of the power supply. It would also require less work to bury a device with a small antenna, compared to a device with a big antenna. A problem in such a solution is the radio module range. If some devices were to be placed on a far corner of a golf course, the devices might not be able to transmit its data to a central node.

4.4.2 Security

This section contains the security considerations regarding unauthorized access, data loss or theft.

The main function of the system is to transmit data wirelessly through a network. Since the data is sent wirelessly, it can be intercepter or interfered resulting in incorrect data.

A problem could be the process of connecting a new device to the network. If a device is connected without any authentication, it could become a problem if a neighbouring network is using the same frequency which could result in the two networks exchanging data unintentionally. A malicious device could also inject data into the system.

Considering the hardware design of a device, it would also be relevant to analyze if the device can be physically reconstructed to intentionally transmit malicious data to the network. Though being more a question of physical security, by preventing access to the device, this could still be relevant.

The major issue in the system is that data is transmitted wirelessly and it is therefore exposed to malicious input. This should be taken into account in the design of a final solution.

4.4.3 User interface

As the main purpose of the system is to transfer data from sensors to an end node, where a user can read and react appropriately, it would be necessary to present this data for the user. Since it is practically impossible for humans to decipher bytestreams, a user interface with a more simple way of presenting the data would be practical, by giving a better overview of the data.

An excessive amount of interactions with the user interface would be unnecessary, as the system would only serve as a way for reading and understanding the collected data.

5. Problem statement

This chapter contains the final problem statement, the requirements of the solution and the reason for these requirements.

5.1 Problem definition

The answer to the initializing problem statement is, based on the analysis chapter, that embedded systems can indeed be utilized to assist in maintaining a golf course. During the interview, the interviewee presented the idea of collecting data about the golf course soil to assist the greenkeepers' tasks. Since this data should be collected from various spots on the golf course, embedded devices could be used to achieve this. This results in the final problem statement:

How can a sensor network and a corresponding protocol be designed for a golf course, so that data can be relayed throughout the network, enabling an endpoint device to receive the information without being within range of all sensors in the network?

Before development of the solution can begin, the requirements found throughout the analysis, will be evaluated and sorted in order to form a requirement specification.

5.2 Requirements

In this section the requirements of the project is sorted based on different priorities.

Requirements are an essential part of developing a system that fulfills its purpose. All requirements can be considered important, but some might be more beneficial for the core purpose of the system. Since this project has a limited time frame it is required to limit the project size correspondingly. By categorizing the requirements it becomes possible to identify and separate the important criteria from the lesser ones.

The MoSCoW (Must-have, Should-have, Could-have and Wont-have) analysis method has been used to sort the requirements found in the analysis chapter. This method is used to ensure that the project will result in a working solution, by finding the core requirements for a working solution. These are the ones listed as 'must-have'.

If the the time-frame of the project is then longer than the time required for developing the core system, other requirements can be considered. These are the ones listed in 'should have' and 'could have'.

Finally the 'won't have' category are requirements left out, by being unrealistic to develop or not relevant for the project. In this project the wont-have category

has been left out, because no requirements met the criteria for that category. The requirements are summarized below:

Must have:

- Correct, wireless transfer of data between nodes.

The core purpose of the system is to transfer data between nodes correctly. Without this, the system would not be able to fulfill its purpose. This also implies that the transferred data will be verified. A communication protocol is needed to ensure that the data is treated correctly.

The amount of nodes can be considered somewhat unclear, but should be enough to cover the entire golf course. In this solution this amount is a minimum of 100 devices, to at least cover the greens of 18 holes, which gives at least 5 devices for each hole.

- Appropriate handling of a disconnecting node.

Since nodes are to be dug down at different places on a golf course, it becomes necessary to ensure that the entire system does not break down if one node were to disconnect. If each node were to be dug up and reset if one node in the whole system got disconnected, the maintenance cost could quickly become too great.

- Sensors.

The nodes on the golf course must collect some data to transfer back to the main node. Based on the interview, there are three types of properties that are relevant to monitor.

Should have:

- A simple process of adding nodes.

The requirement aims to add flexibility both during the installation of the system, and for further configuration thereafter. This would enable the addition and re-positioning of nodes without requiring a restart or re-configuration of the entire system. As the system **must have** an appropriate handle of a disconnecting node, this can be seen as an extension which the system **should have**, which together would give hot-plugging functionality.

- Modular build.

The embedded system will consist of multiple parts, for the different purposes. A component for wireless connectivity, a sensor to monitor the soil properties, a power supply, and finally a processing unit. These parts should be put together in such a way that if a part breaks down, it would be a simple task to replace it.

- Low power consumption.

With low power consumption the final system can run without maintenance for a longer period of time.

Could have:

- Graphical user interface.

The data collected by the system could be processed by the main node and presented through a graphical user interface to give a better overview of all the data collected.

- Separate networks.

If the final system includes the hot-plugging functionality, the individual nodes should stay connected to a particular network and not accidentally interfere another nearby network, by being added to that. This requirement could potentially become very complex, as this is a question of security, which is not the goal of this project.

6. Technologies

There are multiple technologies available to utilize in a solution. This chapter contains information, descriptions and examinations of the considered technologies, as well as definitions used later in the report.

First, the power source for the nodes are examined, and then controller units for the sensors will be described. Secondly, sensor devices are studied to be able to choose an adequate unit for the solution. Thirdly, communication devices are examined to select an appropriate unit to use in the solution.

In the last part of the chapter, network topologies and communication protocols are examined to explore ways to implement a network of nodes transmitting sensor data. The chosen components, network type, and protocol can be found in chapter 8.

6.1 Platforms

The suitable platforms for this project are examined in the following section. These platforms are capable of executing instructions as well as reading or writing to pins available on the board. These pins can be connect to components such as sensors and actuators.

6.1.1 Arduino

Arduino is an open source platform, which makes the software and hardware documentation available to the public. The name "Arduino" covers both the software platform and the range of hardware platforms with boards of different sizes, from the smaller Arduino Nano up to the Arduino Mega. One of the more popular Arduino boards is the Arduino Uno which is a medium sized board, often used by starters [9].

How the Arduino handles or reacts to input is implemented by the user and uploaded to the Arduino board by using the Arduino IDE.

Arduino Uno
One of the most common Arduino boards is the Uno, seen in figure 6.1, which uses the ATmega328 microcontroller [9]. It has 14 digital input/output pins, and 6 analog inputs for connecting different components. Considering specifications, which is shown in table 6.1.1, the Uno is limited on its resources. Therefore it is needed to limit both program and data size, and also the amount of complex tasks.

Arduino Mega
The Arduino Mega is a larger version of the Uno. The Mega has more memory and pins, which makes it better for handling larger programs and amounts of

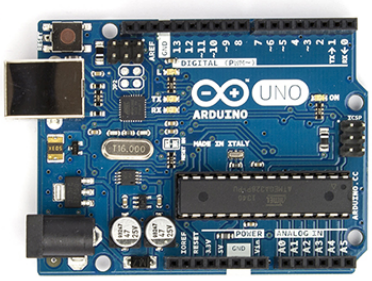

Figure 6.1: The Arduino Uno board[10].

Unit	Arduino Uno	Arduino Mega
Microcontroller	ATmega328	ATmega1280
Clock Speed	16 MHz	
Operating Voltage	5V	
Input Voltage (recommended)	7-12V	
Input Voltage (limits)	6-20V	
DC Current per I/O Pin	40 mA	
DC Current for 3.3V Pin	50 mA	
Analog Input Pins	6	16
Digital I/O Pins	14 (6 provide PWM output)	54 (15 provide PWM output)
Flash Memory	32 KB (0.5 KB used by bootloader)	128 KB (4 KB used by bootloader)
SRAM	2 KB	8 KB
EEPROM	1 KB	4 KB

Table 6.1: Arduino Uno and Mega specifications [9][11].

data, and also allows more components to be connected to the board. Since the clock speed is the same as the Uno, the Mega will not process data faster[12].

6.1.2 Raspberry Pi

Raspberry Pi is a series of single board computers. The Raspberry Pi series contains some powerful controllers compared to other pocket sized processing units.

The Raspberry Pi has a set of pins for connecting to external hardware. There are four different pin types available; two 5v power, two 3.3v power, five ground and 17 GPIO. In figure 6.3 the placement of the pins is shown.

The GPIO, or "general purpose input/output" pins can be used to either send or

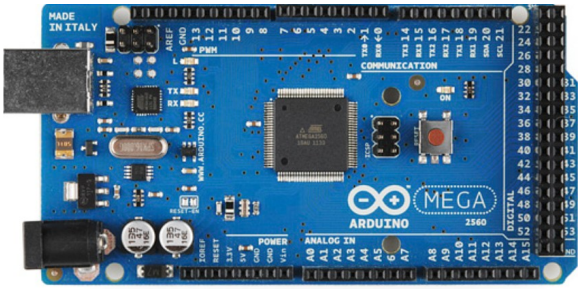

Figure 6.2: The Arduino Mega board[13].

receive digital signals. Unlike the Arduinos it is stricly digital, as it does not have a build in analog to digital converter. If analog input processing is needed, this will have to be done externally.

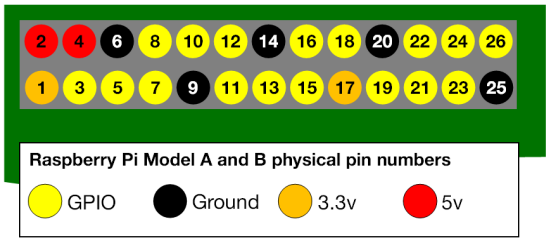

Figure 6.3: Raspberry Pi pins on model A and B.

Because of the computing power and memory of a Raspberry Pi, a Linux OS is usually installed on the Raspberry Pi[14]. A Raspberry Pi also has a GPU, video output, and USB port.

Raspberry Pi B+

In this subsection a Raspberry Pi B+ will be described as these were available to use in the project. The different models usually differs on CPU speed and memory.

The Raspberry Pi B+ specifications are shown in table 6.2. Futhermore the B+ also contains HDMI video output and Ethernet connectivity. A micro SD card is used as main storage a micro, which contains the OS.

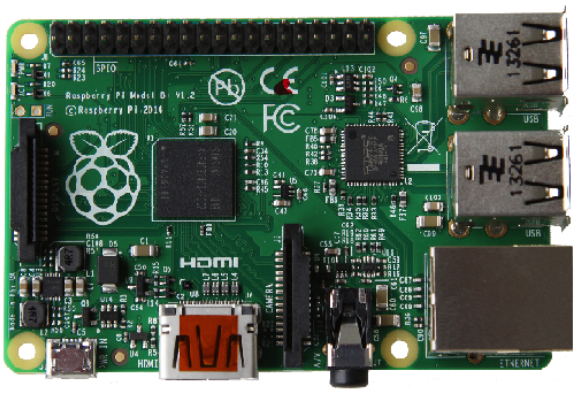

Figure 6.4: Raspberry Pi B+[15].

Microcontroller	Broadcom BCM2835
RAM	512MB
Extended GPIO Pins	40
USB Ports	4
Clock Speed	700 MHz

Table 6.2: Specifications for Raspberry Pi B+[16].

6.2 Power supply

Each node in the network will require power in order to function. There are multiple ways of powering the nodes, each with their own strengths and weaknesses. Some of the relevant possibilities are the use of power cables, solar panels, and batteries. Each of these possibilities are discussed throughout this section.

Power cables

Power cables could be used as a steady power source for network nodes. The cables would be dug down throughout the golf course whereof each node would get connected. This solution is ideal since every node in the network will always have their required power at hand, so running out of power will never be an issue. The power cables are therefore a solid long term way of providing power.

The requirement of the final overall solution that states that the nodes must communicate wirelessly becomes redundant if power cables were to be utilized. If the golf course would need to be dug up in order to lay down the power cables, cables for communication could just as well be laid down to ensure reliable communication.

The permanent aspect of the power cables may not be a strength in the long run. The areas of the golf course that are of interest to the greenkeepers might change during the season. The power cable solution would not allow the re-positioning of the nodes since they are dependent of the cables. This also puts constraints on the scalability of the final solution, since the addition of extra nodes to the network requires that the golf course gets dug up again.

Solar panel

Solar panels could be utilized to power the nodes of the network. Each node would be equipped with its own small solar panel and be self sufficient. This solution, just as the power cables, will not run out of power. The solar panels also gives the nodes portability/flexibility, so they can be moved around the golf course if needed, as well as adding scalability for not requiring to the golf course to be dug up.

Since solar panels need the sun for generating power, the panels need to be above ground and cannot be dug down together with the sensor. This is a weakness since this puts the solar panels at risk of golf balls, golf players, and lawnmowers. This means that the solar panels puts restrictions on the node placement, because it needs a placement where it can generate power from the sun, but at the same be sheltered from harm.

Batteries

Equipping each node in the network with its own battery is another possibility. Utilization of batteries adds portability/flexibility to the nodes in the network, as opposed to the power cable solution. Nodes with a battery can be moved around the golf course if needed, instead of being dependent of where the power cables have been dug down. This also adds scalability to the final solution since the extra nodes can be added to the network without needing to dig up the golf course.

Something to take note of is the lifespan of the battery. The power cables will keep delivering the required power as long as needed. The same goes for the solar panels. Batteries on the other hand, cannot keep up with the two alternatives in this aspect. When the battery runs out, it will be required of a greenkeeper to manually replace it. This means that the usage of batteries adds a constraint to

the entire system about power consumption, which would have to be taken into consideration when designing the final solution.

6.3 Sensors

To acquire relevant data from the golf course, and to enable processing of this data in a digital domain, a sensor is needed. The sensors must be capable of measuring data, and returning a value, either as a digital or analog signal that can be processed by a computation unit. As multiple sensors are needed, the price point is considered relevant, as it will cause the overall cost of the solution to grow.

There exist several kind of sensors, such as soil pH sensor, soil density sensor, temperature sensor and many more. In this project only the soil moisture sensor will be covered, since adding a different sensor would be trivial.

6.3.1 Moisture Sensor

To obtain information about the moisture of soil on the golf course, a moisture sensor is needed. The moisture can change relatively quickly depending on rain and and type of soil. The moisture sensor used in this project is a Arduino Soil Hygrometer Detection Module Soil Moisture Sensor. The sensor operates with a voltage of 3.3V to 5V. It has dual output, both digital and analog, with analog being the most accurate. The digital output is in the form low(0) and high(1), while the analog output range from low(0V) to high(5V) [17].

When the output is high, the moisture content of the soil is high, while a low output means a low moisture content. Calibration is needed when there are several sensors which should provide the same range of moisture. Figure 6.5 shows the moisture sensor, with the fork being the actual sensor, and the circuit board being the converter and calibrator.

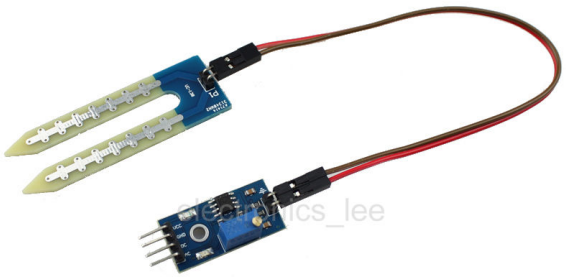

Figure 6.5: Soil moisture sensor.

6.4 Communication devices

In the following section, different types of communication technologies and devices relevant to the project are examined.

6.4.1 Bluetooth

Bluetooth is a wireless technology standard designed to transfer data over short distances[18]. Bluetooth is often used in phones and computers, but are suitable for embedded systems requiring wireless transfer of data.

The number of devices that can be on a Bluetooth network is almost unlimited, and the the built-in interference reduction makes the technology usable for the solution[18]. Bluetooth devices have to be paired in order to exchange data, which makes it hard to create a hot pluggable network of devices, as all devices communicating will have to be paired together, and not just added to the network. This, and the short maximum range of 100m[18], could provide problems with using Bluetooth.

6.4.2 XBee

XBee is a radio module by Digi. Multiple versions of the XBee modules exists, with differences in power usage, range, built-in/external antenna, and some

with processors [19]. XBees often use the ZigBee protocol. The ZigBee protocol is made for low-power devices[20].

XBees are fast and predictable, making them a good choice for the solution in this project. They can cooperate with the Arduino platform using the serial pins. There also exists special XBee shields made for Arduino.

XBees are quite expensive, which makes them unfit for this project. If the solution had to interface with other devices using the ZigBee protocol or XBee modules they could be implemented.

6.4.3 nRF24L01 Transceiver

The nRF24L01 transceiver module is a radio module used for exchanging data between modules of the same frequency. As with most of the XBee modules, the RF24 uses the 2.4GHz band, which is license-free in the entire world[21]. It is well documented, and multiple libraries exists for using the device with Arduino.

nRF24L01 is a half-duplex device, which means it can either only transmit or receive data at any given time. An example of a full-duplex devices could be a telephone where both devices can transmit and receive data simultaneously.

The nRF24L01 is cheaper than the XBees, and has shorter range and is potentially slower, depending on the communication protocol.

The full specifications can be seen at[22]

6.5 Networks

There are several terms in the network domain, and this section will contain descriptions of networks and network theory, as well as definitions used in this report.

The implementation of a communication network is necessary, as the sensor system is supposed to reduce the amount of work for measuring the soil properties on the golf course. The current method of measurement is time-consuming, because of the travel time used to cover the designated area of the golf course. A network implementation of the sensor system can transmit the data gathered from each sensor to an end point.

A computer network is a collection of computers and devices connected so that they can share information and services [23]. Devices and connections in computer networks can be modeled with graphs, and therefore the network terminology used will be similar to graph terminology.

A topology is any arrangement of nodes that can be connected with edges [24, p. 628]. A network topology is the arrangement of the nodes, using edges as the structure. There are different types of network topologies and following are some examples:

- Ring
- Tree
- Star
- Mesh

Examples of these can be seen in figure 6.6.

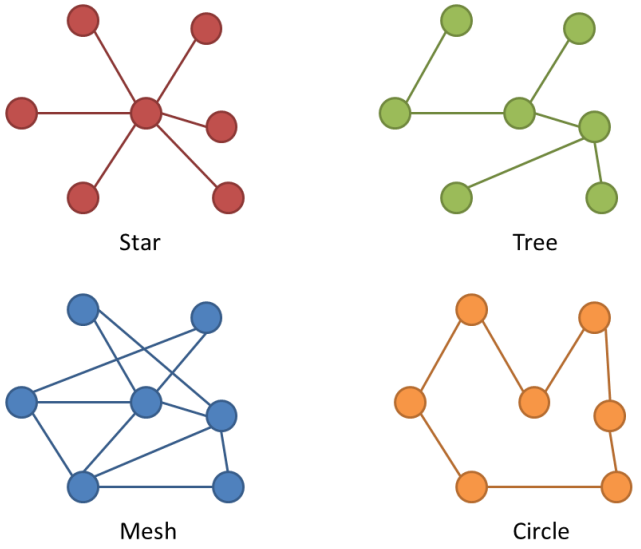

Figure 6.6: Network topologies.

The star topology, seen in figure 6.6, has one main node that the other nodes are directly connected to. An example of a star topology network is Wi-Fi, typically with a wireless router to which other devices are connected to gain network access. The wireless router will handle the network communication and redirect the information to the correct device. A limitation of the star network is that all nodes must have a direct connection to the main node, and should therefore be within transmitting range of the main node.

A tree topology, seen in figure 6.6, utilizes a main node, called the root of the tree. The devices in the network do not necessarily connect directly to the main node, but rather connect to another node that relays to the main node. This can repeat over multiple levels, so that information is relayed through several nodes, before reaching the main node. The tree network has a fixed node structure, and the

Topology	Pros	Cons
Ring	Limited to two directions of communication	Requires some ring formation of the nodes in the network Information delay through the network
Tree	Reach through internode communication from leaves to root	Cannot contain loops
Star	Fast, direct communication to main node	All nodes must have transmit reach covering the main node
Mesh	Reach through internode communication Not critically sensitive to node failure	Information delay through the network

Figure 6.7: Topology strengths and weaknesses.

nodes will relay and route the information towards the destination [25]. It has a reach advantage over the star topology, as data of nodes, not directly within reach of the main node, can still be delivered to the destination node.

The ring topology, seen in figure 6.6, connects all nodes in a circuit where the path passes through all nodes exactly once. This topology does not require a central node to control the data communication, and will, like the tree topology, relay the data through a certain chain of nodes before it reaches the destination node. It is vulnerable, while there is only two directions to communicate, and if a connection is lost, there could be a block in the data flow since the data must be sent through all nodes to get to the destination. If any edge is removed from the circuit, the remaining graph is a line, resulting in a chain of communication.

Another topology is the mesh, also in figure 6.6. There are two kinds of mesh networks: The full-mesh and the partial-mesh networks. A full-mesh describes a network where all the nodes are interconnected, similar to a fully connected graph. In this topology, there will be no redirecting or relaying, but just a direct connection to the destination node, regardless of the destination. A partial mesh is also a mesh network, but does not require all nodes to be connected, so that it's similar to a tree topology, but cycles can occur. The partial-mesh must then support relaying of data, to transfer data from any node to the destination node.

A summary of the network topology strength and weaknesses is show in table 6.7.

6.6 Communication protocols

There are multiple methods for communicating in a network, and such a procedure is called a protocol [26]. A protocol is a set of rules regarding, for example, the information format, procedures for sending and receiving, error discovery

and handling, and it can be implemented in both hardware and software.

This section contains descriptions of multiple wireless network communication protocols that can be used in this project. These are considered and used as inspiration for the design and implementation of a protocl. First flooding protocols, followed by routing protocols.

There are many protocols and multiple groups of protocols, therein routing and flooding. Routing will transfer the information to the destination node through a determined route, whereas the flooding method will notify all nodes within reach to distribute the information forward, and this will repeat until all nodes has transmitted the information, and hence the destination node also has received the information.

Main node in this section is defined as the device that handles the requests or sends the initial packet. Packets are data sent though out the network. A packet can contain data to be delivered, or metadata to keep track of the packet, such as the sender ID of the packet.

6.6.1 Flooding protocols

Flooding in networks is a protocol used to deliver data throughout a network. There are two kinds of flooding: uncontrolled flooding and controlled flooding[27].

Uncontrolled flooding is the broadcasting of a packet, with a particular recipient, but without a particular route to reach the destination. Each node receiving the packet will repeat it to all of its neighbors, and they will repeat it again to theirs. This causes the packet to reach all nodes in the network, but it can cause communication loops[28].

Controlled flooding is flooding protocols with filters or other measures to make it more reliable or efficient than the uncontrolled, hence avoiding cycles and broadcast storms[27]. Examples of such algorithms are Sequence Number Controlled Flooding (SNCF) and Reverse Path Flooding (RPF). With SNCF, each node has its own address that is delivered with each data packet it transmits. If a node receives a packet multiple times with the same unique id, the packet is discarded, so only one instance of the packet is transmitted. In RPF, a node will forward a copy of a received packet, no matter the origin. RPF expects that the original transmitted packet will be copied by the nodes in a path to the destination and eventual delivered[29].

Advantages of flooding, is that if there exists a route from the source to the destination, a packet will be delivered, and up to multiple times, depending on the implemented protocol. Since some flooding algorithms utilizes every route in the network, it will also transfer packets through the shortest path[28].

Disadvantages can be that the bandwidth is wasted, since flooding is a very costly algorithm as it utilizes every route. Packets can be duplicated in the network, further increasing bandwidth load. Duplicate packet may loop in the system forever, if no prevention is made [28].

6.6.2 Time division multiple access

Time division multiple access(TDMA) is an access protocol that divides a single channel into smaller time slots. Each timeslot is assigned to a single node, while every timeslot, are on the same channel. Each timeslot is active a short period of time, allowing the assigned node to transmit, before the next node in the queue get time to transmit[30].

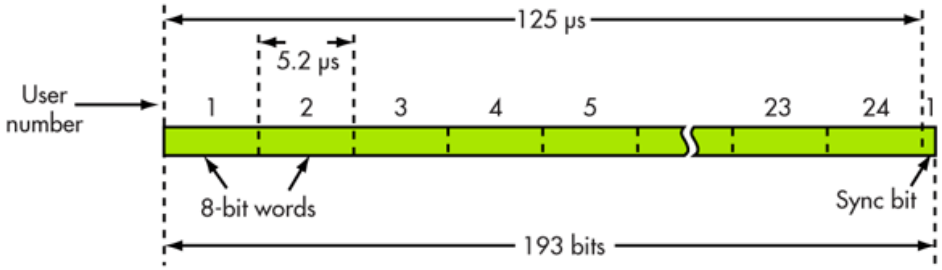

Figure 6.8: Illustration of the TDMA access method in the T1 protocol[30].

In figure 6.8 it is seen how the channel is split up into 24 smaller pieces. After the queue of timeslot there exists a bit, called the "Sync bit" as seen on the figure, that is used for synchronization. Dividing the channel into smaller parts have little effect on the nodes transmission, because the shift between each time slot happens so fast. This gives the illusion of each node communicating with no interruptions, even though each one is only assigned 1/24 of the total bandwidth on the channel.

TDMA can be used for any system that require several device to use the same channel, where interference can be a problem. For example when using radio transmitters as they can cause a lot of interference, and that is in one of the areas TDMA can be used[31].

6.6.3 Dynamic Source Routing protocol

The Dynamic Source Routing protocol (DSR) is a simple routing protocol designed specifically for use in wireless ad-hoc networks. With DSR, the network is completely self-organizing and self-configuring, requiring no administration or existing network structure [32]. As nodes can be added and removed as desired, the protocol automatically determines and maintains the routing of the

packets through the network. Since the number or sequence of nodes in a network may change at any time, the topology may be quite rapidly changing. DSR works on demand, allowing the routing of DSR to scale automatically, affecting only nodes that is needed. The protocol provides a highly responsive service to ensure successful delivery of data packets where nodes may be moving around, or other changes in the network occur[32].

The DSR protocol is composed of two main mechanisms that allow the discovery and maintenance of the routes in the network.

Route discovery is the mechanism by which a node S wishing to send a packet to a destination node D, obtains a route to D. Route Discovery is used only when S attempts to send a packet to D and does not already know a route to D.

Route maintenance is the mechanism by which node S is able to detect, while using a route to D, if the network topology has changed such that it can no longer use its route to D, which is known as a broken route. When Route Maintenance indicates a route is broken, S can attempt to use any other route it happens to know to D, or it can invoke Route Discovery again to find a new route for subsequent packets to D. Route Maintenance for this route is used only, when S is actually sending packets to D.

DSR is a routing protocol where packets carry a header, an ordered list of nodes which is the route. This explicit use of routing allows the sender to select and control the route for the packets it sends. Routing allows for load balancing, since the sender can create different routes out through the network, avoiding high throughput on few nodes. It is also a guarantee that the routes used are loop-free, since a generated route never use the same node twice. By including this route in the header of each packet, other nodes forwarding or overhearing any of these packets can use this information for future use[32]. When a node overhear a packet with a route, the route is stored locally on the node. The node can then use the route to forward messages, if its old route is unavailable.

6.6.4 Ad-hoc On-Demand Distance Vector Routing

Ad-hoc On-Demand Distance Vector(AODV) routing algorithm is a routing protocol designed for ad-hoc networks. It is an on-demand algorithm, meaning that it builds routes between nodes only as necessary by main nodes [33].

AODV is a network where only the nodes within reach of a newly added node is affected by the addition. The transmission route in AODV is managed so that only nodes in the direct route are active. The protocol can determine multiple routes between a main node and a destination, but only a single one is implemented the route is not necessarily the shortest [33].

A disadvantage of AODV is that if a single route breaks, for example due to a

defect node, it is not possible to know whether other routes exist. A new route discovery in AODV is to be carried out before knowing if there exist a route. If a link between nodes is broken, and it does not affect an ongoing transmission, no notification back to the main node occurs.

When a node is about to forward a packet to a specified destination, it checks its routing table to determine if there currently exists a route to the destination. The routing table is a table over every route though the node to any other, already discovered, destination. If there already exist a route, the packet will be delivered to the next node in the route, repeating until it arrives at the correct location. If there does not exist a route, the node will initiate a route discovery process [33].

A route discovery process in AODV begins with the main node creating a Route Request (RREQ) packet. The packet contains the main node's unique ID and the destination node's unique ID. The packet is then transmitted from the main node to it's neighbours. The unique ID of each node is then appended to the packet, to ensure the same node is not transmitting more than once. The destination ID is to ensure that the nodes in the route knows when the destination node is reached. A simple AODV network can be seen in figure 6.9, where S and D represents the main node and destination. It is visualized how the route discovery is invoked using the RREQ packet [34].

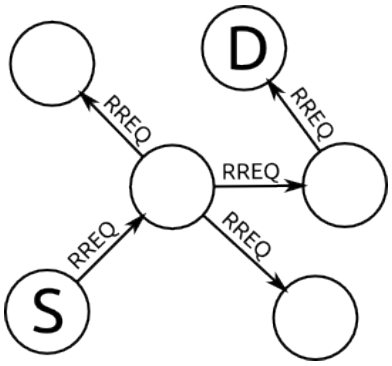

Figure 6.9: Illustration of the AODV route discovery.

6.6.5 Better Approach To Mobile Ad-hoc Networking

Better approach to mobile ad-hoc networking (B.A.T.M.A.N) is a routing protocol for ad-hoc mesh networks, based on AODV. B.A.T.M.A.N is said to solve some of the more typical problems with the classical routing protocols. Some of the problems are that some networks are unstructured, can be based on an inherently unreliable medium and dynamically change their topology[35].

When using the B.A.T.M.A.N algorithm, the approach is to share the knowledge about the most reasonable path between nodes in a network to all of the partic-

ipating nodes. Each node in the network only accounts for the best path to all other nodes in the system. This discards the need for a global knowledge about local topology changes. In addition, B.A.T.M.A.N has an event-based, but timeless, flooding mechanism that prevents the occurrence of loops in the network.

Each node in the network transmits a broadcast message(called OGMs or originator messages in B.A.T.M.A.N) to inform every node within range about its existence. The nodes within range, then re-broadcast the OGMs, to nodes within their range, about the existence of the original initiator of this message and so on. This can result in the network being flooded with OGMs. OGMs are small, the typical raw packet size is 52 byte including IP and UDP overhead[35]. OGMs contain at least the address of the originator, the address of the node transmitting the packet, and a sequence number.

OGMs that follow a path where the quality of wireless connection is poor will suffer from packet loss or delay on their way through the network. Therefore OGMs that travel on good routes will propagate faster and be more reliable[35].

In order to tell if a OGM has been received once or more than once it contains a sequence number, given by the originator of the OGM. Each node re-broadcasts each received OGM at most once and only those received from the neighbour which has been identified as the currently best next hop (best ranking neighbour) towards the original initiator of the OGM.

This way the OGMs are flooded selectively through the network and inform the receiving nodes about other node's existence. A node X will learn about the existence of a node Y in the distance by receiving it's OGMs, when OGMs of node Y are rebroadcasted by its neighbours.

The algorithm then selects this neighbour as the currently best next hop to the originator of the message and configures its routing table respectively [35].

6.6.6 Protocol comparison

Table 6.3 shows comparisons between the protocols discussed in this chapter. The four different protocols is compared in what route metric they are using, if they are loop free, support of load balancing, how reliable they are, and the estimated throughput.

One big difference to notice in table 6.3 is that TDMA is the only protocol with load balancing. None of the other protocols have no such feature, which can result in some nodes in the network handling a heavy load, ultimately creating a bottleneck.

B.A.T.M.A.N is the protocol which scales the best. The other have troubles keeping up as more nodes are added. In addition B.A.T.M.A.N and AOVD are the two

	Ad hoc	Route metrics	Loop Free	Load balancing	Reliability	Throughput
TDMA[36]	Yes	Routes ensuring guaranteed bandwidth	Yes	Yes	High	Decreases as more nodes are added
DSR[37][32]	Yes	Source routing	Yes	No	High	Decreases as more nodes are added
AOVD[37]	Yes	Fastest & shortest path	Yes	No	High	Poor for more than 20 nodes
B.A.T.M.A.N[35]	Yes	Fastest & shortest path	Yes	No	High	Good - scales well with more nodes

Table 6.3: Protocol comparison.

protocols with the fastest routing algorithm as they always utilize the shortest path.

The information gathered in this section will ensure a good foundation for designing a protocol for a network specified to work on a golf course.

7. Data transmission

In this chapter the theory and considerations on data transmission are described. First, exponential backoff is described, which helps reduce interference and collisions. Next are techniques to verify that data transmitted are not received as corrupted, obfuscated or in any other way different from the actual data transmitted.

7.1 Exponential Backoff

The technique *exponential backoff* is a method used for reducing interference or collisions. The technique works by incrementing the interval between sending packets which will increase the probability that the packet will get transferred without any other node interference.

A random interval is needed as the nodes need different intervals. The random interval will improve the chances that one of the sensors will send while the other is waiting. Without the random interval the two nodes could collide on the first back off, and both nodes will then back off with the same amount, resulting in another collision. This can keep occurring since the two nodes keeps backing off with the same interval.

$$E(c) = 2^{c-1} \tag{7.1}$$

The equation used to calculate the maximum backoff interval in milliseconds as stated in equation 7.1, where c is the current number of collision in transmission between two or more nodes. There will be added a range from 1 to the calculated maximum backoff, to ensure a range to pick a random number from. Using this equation to calculate the backoff, the delays can be seen in table 7.1. Figure 7.1 shows how exponential backoff is exponentially increasing.

Attempt	Back off range	Best case	Average case	Worst case
1	1 to 1	1	1	1
2	1 to 2	1	1.5	2
3	1 to 4	2	3.5	7
4	1 to 8	3	7.5	15
5	1 to 16	4	15.5	31
6	1 to 32	5	31.5	63
7	1 to 64	6	63.5	127
8	1 to 128	7	127.5	255
9	1 to 256	8	255.5	511
10	1 to 512	9	511.5	1023
11	1 to 1024	10	1023.5	2047

Table 7.1: Exponential backoff in milliseconds.

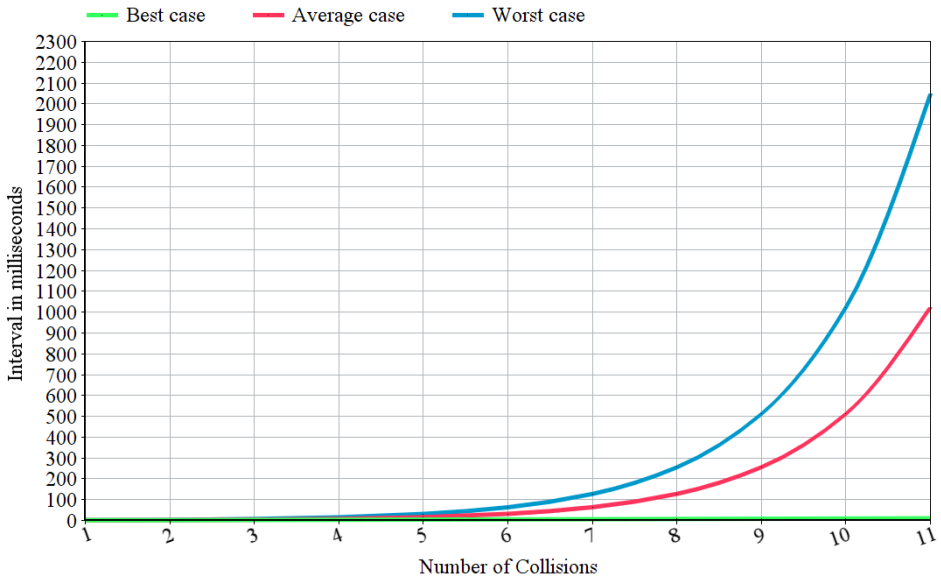

Figure 7.1: Exponential backoff in milisecounds.

7.2 Data verification

Wireless communication is prone to noise and interference[38]. To be able to discern correct packets from incorrect packets, a detection method is necessary. There are multiple ways to verify the integrity of a packet and this section contains a description of some methods and the reasoning behind the solution.

7.2.1 Definition and types

A checksum is a unique value that can be used to verify that a message is received correctly [39]. The checksum is based on all parts of the message and unique to the length of the checksum. A single bit checksum can only distinguish between two messages, and is not able to verify with more than 50% probability, as an error could affect the transferred message, so that the checksum calculation still produces the same checksum. Checksums can variate by multiple properties, for example by how it is calculated, or by its length.

Digit sum

A simple algorithm for producing a checksum is to use the digit sum and store it in a byte. For example, the digit sum of 1337 is $1+3+3+7=14$. This algorithm is efficient, but has several flaws, therein the uniqueness of the checksum of only 256 values and that two bit flips could cancel each other out and cause the same checksum. The error detection level is regarded as too small, and this algorithm will not be used in the protocol.

MD5

MD5 is a widespread checksum algorithm. It produces a 128 bit checksum, which is 16 bytes. MD5 is designed to run efficiently on 32 bit processors, which is a disadvantage on the Arduino boards used in the solution, since these posses an 8 bit processor. The 16 byte checksum uses half of the possible payload of the nRF24L01, which is too much. The MD5 algorithm is therefore discarded as the checksum generator for this project [40, p. 308].

Cyclic redundancy check

Cyclic redundancy check is a checksum algorithm, hereafter referenced to as *CRC*. CRC exists in several forms, based on the parameters, such as the checksum size and internal algorithm parameters. CRC can be used to detect either one or more 'single bit error' and burst errors [41, p. 31]. Due to the small check-

sum size and the error detection properties, CRC is chosen as the checksum generator.

7.2.2 Computation of a CRC checksum

In short, the CRC executes a division of the message on a polynomial, and the remainder is used as the checksum. The CRC algorithm requires both sender and receiver to agree upon a generator polynomial in advance.

A message M is viewed as a polynomial with binary coefficients, by treating each character in M bit-wise. This makes for a relatively large number, but the decimal equivalent is irrelevant for the computation. The length of M in bits is called m, which as exemplified in table 7.2 is 24.

String	CRC
Binary	01000011 01010010 01000011
Decimal	4,411,971
Polynomial	$x^{22} + x^{17} + x^{16} + x^{14} + x^{12} + x^{10} + x^6 + x^1 + x^0$

Table 7.2: String equivalents example.

A generator polynomial of degree r, generates an r bit checksum. The generator polynomial is the dividend which generates a quotient and a remainder for the message. Some polynomials are prone to creating multiple similar remainders, while others generate mostly unique remainders and it is therefore advantageous to use already developed, tested and implemented polynomials [42].

The execution of the division itself is done by binary long division of the polynomial representation of the message by the generator polynomial. This is done without carries and means, that any bit x^k is not affected by the bits x^{k+1} or x^{k-1}. This kind of division propagates to be an *exclusive or* function, as seen in table 7.3, applied from left to right on all bits within the length of the polynomial through the message and only when the leftmost bit in the message is 1. To illustrate the algorithm, an example of the division can be seen in figure 7.2.

$\pm$	0	1
0	0	1
1	1	0

Table 7.3: Binary addition and subtraction without carries.

The algorithm for generating a transfer message is the following, when a message M, a size r of the checksum and a polynomial G is decided:

1. For degree r of G: append r zeroes to the lower end of M, s.t. $M + R = Mx^r$

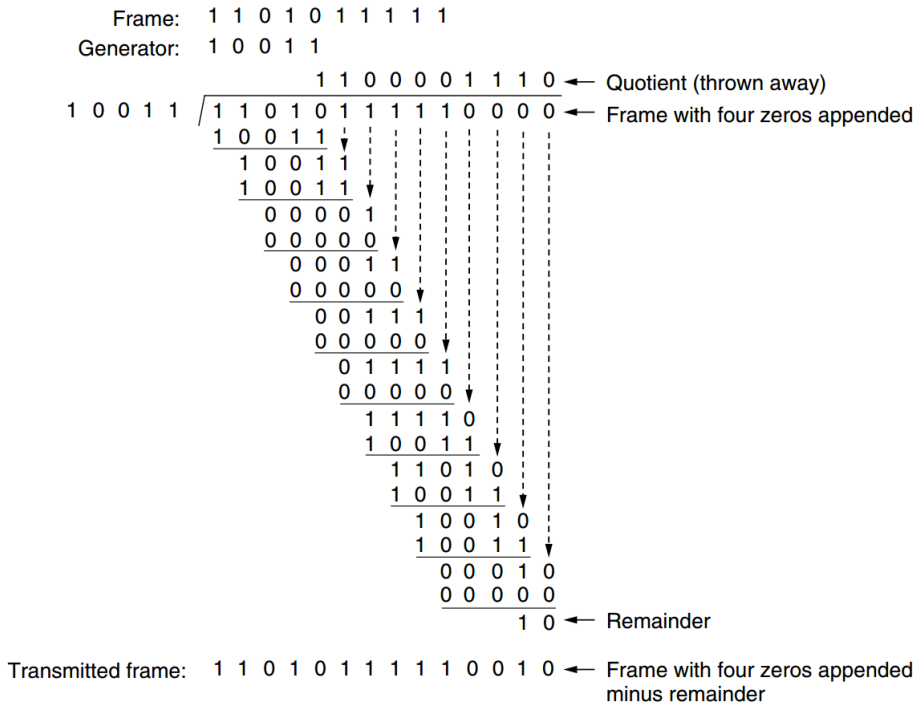

Figure 7.2: An example of binary long division [43], where "Frame" equals message.

2. Divide Mx^r by G, using binary division
3. Subtract remainder from Mx^r using binary subtraction, resulting in transfer message T

$$Mx^r - QG = R \tag{7.2}$$
$$Mx^r - R = QG \tag{7.3}$$
$$Mx^r - R = T \tag{7.4}$$
$$T = QG \tag{7.5}$$
$$\tag{7.6}$$

Result ($Mx^r - R = T$) is the transfer message, made by the operations $(Mx^r)/G = R$ and $Mx^r - R = T$.

The equation for the checksum is shown here. M minus some quotient/multiple of the generator polynomial G leaves a remainder R, as seen in equation 7.2.

8. Design

This chapter contains the design choices based on the analysis, requirements and the technologies previously examined. The decisions made in this chapter forms the foundation of the implementation process.

First, the overall intentional system design is described. Thereafter, the components used for the sensor nodes are selected, herein platforms, sensors, and communication devices. Then, the network structure and utilized protocol for the solution are described.

8.1 Design intention

This section contains the description of the intended design and usage of the system. The design is based on the requirement specification in chapter 5.2. The core function of the solution is to transfer data wirelessly from several sensor nodes to a main node, and to be used by greenkeepers.

The human aspect has requirements regarding aesthetics and user interface design, but that is secondary to the task of gathering data. This section will mainly consider the functional requirements to develop a functioning system. Nevertheless, a user must be able to utilize the system, and the main characteristics of the solution will be described in the next section.

8.1.1 Workflow

As the purpose of the solution is to support in monitoring a golf course, the solution should make the task less time-consuming. The intention is that the data should be gathered and updated on demand, so that the properties of the golf course is sufficiently new for the greenkeepers to use.

An on-demand solution has an advantageous property: The system components can save energy by not updating data unnecessarily, and the user is able to evaluate the age of the last measurements and decide whether to request new data or decide if the previous is sufficient. The on-demand solution, however, relies on the time required to gather the sensor readings to be useful. The current reference point is the time it takes to manually check the status of the golf course. Therefore, the data collecting time should be strictly less than time used to do it manually.

A fully automated solution means, that the system could be set to gather data periodically, so that sensor readings are available once or multiple times during a given time frame. This ensures that the data is always up to date. An automated solution, however, has the disadvantages that the system could waste energy on readings that are not needed, or not having readings available when needed, if the time frame is incorrect.

The solution is designed to function according to the following procedure:

1. The system user will use an simple interface to demand readings from the sensors in the network.
2. The main node responds to the demand by broadcasting a request signal in the network.
3. The network nodes receive the request signal, gather data from their sensors and send it back to the main node before creating a new request signal to broadcast.
4. The main node collects all data and appends each reading to the respective origin node in a data sheet made available to the user.
5. The user can evaluate data and take action based on the results.

The solution is deemed on-demand, because of the flexibility it provides. It could also be automated, by simulating a user request at a given time, in addition of letting the users request readings whenever desired.

8.1.2 Data handling and storage

The properties measured by the sensor nodes are the data used in the solution. The solution will monitor the status of the determined properties of the golf course. The observed data needs to be stored and processed to be usable for the greenkeeper in the process of maintaining the golf course. Data from previous requests should be accessible later, making it necessary to store received data.

There exists several methods to request and receive data. One method could be to request data from a single specific node in the network, evoking the nodes along the branch to the destination. The other method is to request data from every node at once, evoking the entire network.

In addition, because of the limited amount of memory available in the sensor nodes, it is not possible to save data about every other node in the system as well as storing all sensor readings on each node. This restricts the historical data to the amount of readings accommodable on the sensor nodes.

Storing the data on the main node, requires a sufficient amount of memory to be able to save a significant history of data. Handling and representing the data requires more processing power than the Arduino is capable. A computer, in the form of a Raspberry Pi, is capable of fulfilling the role as the main node.

The data to be monitored, is the moisture in the ground. This choice was made during chapter 6.3.

The task of the sensor nodes in the system is reading its sensors and transfer the data to the main node. All storage and processing of data will be handled on the main node. The stored data can be used for statistics and other purposes to further optimize the golf course maintenance, but that is considered beyond the scope of this project.

To access this data, a user interface is necessary. The user interface is presented as a website, and will be accessible from any platform by connecting to the main node using a browser.

8.2 Components

Chapter 6 examined the various components usable in the nodes. This section contains the decisions made regarding the components.

Arduino - sensor nodes

Both Arduino Uno and Mega are used in the solution, as they qualify regarding specifications, and due to their availability to the project group. The Mega boards are not faster than the Uno boards, but can contain more program code, more data and have more pins available. While the solution should be runnable on the Arduino Uno boards, the solution is restricted to the Uno's capacity, extra memory and pins on the Mega boards are superfluous. The Mega boards are still used in the project, not because of the extra functionality, but to enable the project group to test a larger system of nodes.

Raspberry Pi - main node

The Raspberry Pi B+ is used in the solution as the main node, based on the need to store, handle, and present data. This device has the responsibility of displaying the user interface, as well as processing and presenting the data received from the sensor nodes, as it has more processing power than the Arduinos.

nRF24L01 Transceiver

The communication device chosen for this project is the nRF24L01, because of its documentation, and low power consumption. The transfer speed of the module is sufficient to send the small data packets, which makes this module suitable for the solution.

The nRF24L01 module contains multiple features for detecting packet loss. These includes checking hashes and sending acknowledgments[22]. These features

makes it harder to replace the radio modules in the nodes, as they are platform specific to the nRF24L01. The analysis examined the possibility of using different radio modules in case some requirements change, which is not possible if using platform specific features.

These features have been disabled, which means that the radio packets does not contain a checksum, nor does it send the default acknowledgments when packets arrives. The speed and output power have also been decreased, so the module will transfer slower, and have increased range.

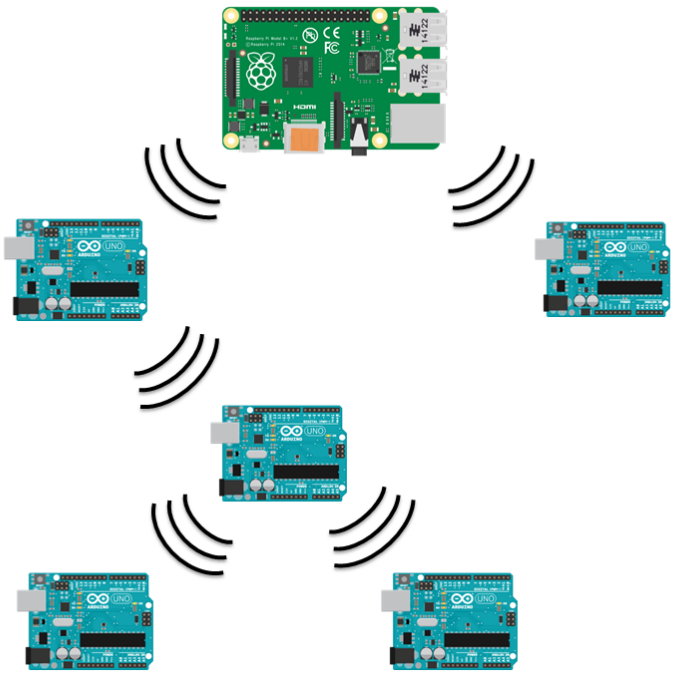

Figure 8.1: Raspberrry Pi main node and Arduino Uno sensor nodes wirelessly connected in a tree.

Setup

With the components chosen and the data sheet for the nRF24L01 examined [22], the sensor nodes will be assembled as seen on figure 8.2. The full system will consist of a single Raspberry Pi as the main node and multiple Arduinos as sensor nodes. An example of the full system can be seen in figure 8.1.

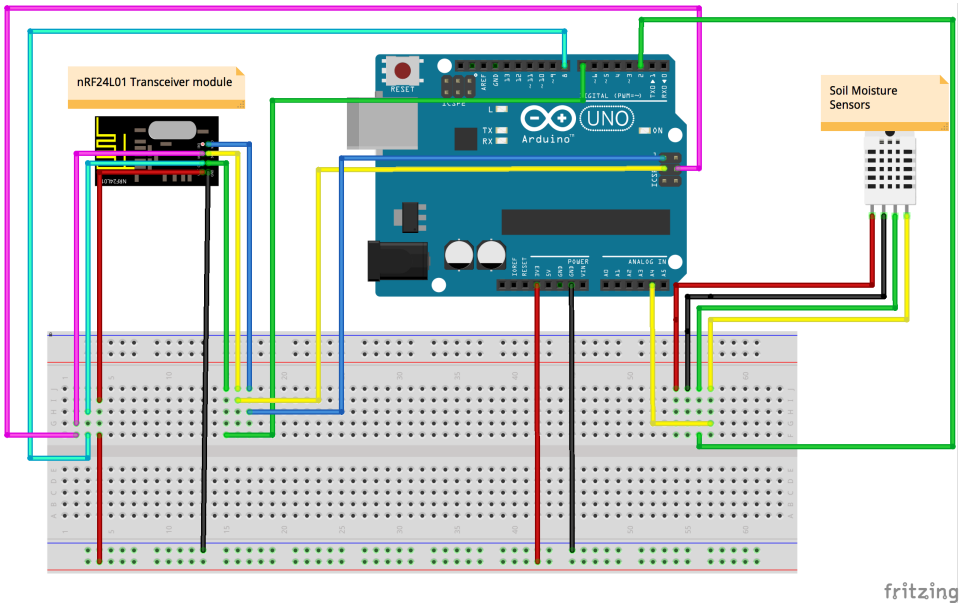

Figure 8.2: Arduino connected to sensor and radio module.

8.3 Protocol communication

To transmit the data from the sensor nodes to the main node, a protocol will be implemented. The protocol is built to be hardware independant, and should thereby work on other components than the ones chosen for the solution.

The protocol to be developed for the solution will be based on the networks examined in chapter 6.5 and protocols examined in chapter 6.6. There are a number of considerations to be made to develop a protocol that statisfies the requirements. Because the sensor nodes are battery driven, most of the choices are done with power consumption as a priority, to increase the battery longevity.

A restriction of the radio modules is their relatively short communication range. The reach of the system can be extended by introducing relaying, so that a sequence of nodes can interconnect and transfer data through the branch in the tree. Therefore, relaying must be supported by the protocol as stated in chapter 5.2.

The radio modules selected for the solution are half-duplex and not capable of full-duplexing, and this limitation needs to be addressed in the development of the protocol.

For power consumption and complying to the usage, the network nodes are controlled by the main node, and the sensor nodes will only transmit data on demand. There is no requirement for real-time monitoring of the golf course, making it difficult to define a data transfer interval that matches both power consumption as well as the need for information. Hence, an on-demand request for data is be sufficient.

8.3.1 Packet Transfer

The solution is developed to gather sensor data from all nodes in a network, and this subsection will contain the considerations regarding choice of transfer method and the according topology.

As the sensor readings are required on-demand, there must be at least two types of communication. The first being the data request; an outgoing signal from the main node, notifying the sensor nodes of the data demand. This is a broadcasted packet, as every node is targeted as recipient. The second type is data response, as the sensor readings needs to be returned to the main node for storage and representation.

Sending individual requests is not appropriate as routing tables would use a significant amount of the limited memory of the network nodes, if the addressee is a sufficient amount of nodes away. In case of disconnected nodes, the routing

tables must be reconfigured, to make sure packets are relayed in a path that actually exist and is within range. This would accordingly require nodes to somehow broadcast that they are still alive, and thereby consume more energy in addition to more memory usage.

The protocol used in the solution is partially based on the controlled flooding protocol, as described in chapter 6.6.1. This protocol is a method where the data request can be transmitted by flooding the network, without the need of sending a tailored request to every node in the network.

The controlled flooding has a low memory impact and thus best suited for the solution. The flooding protocol will create a tree which can be utilized to create a return path for the sensor readings of the sensor nodes.

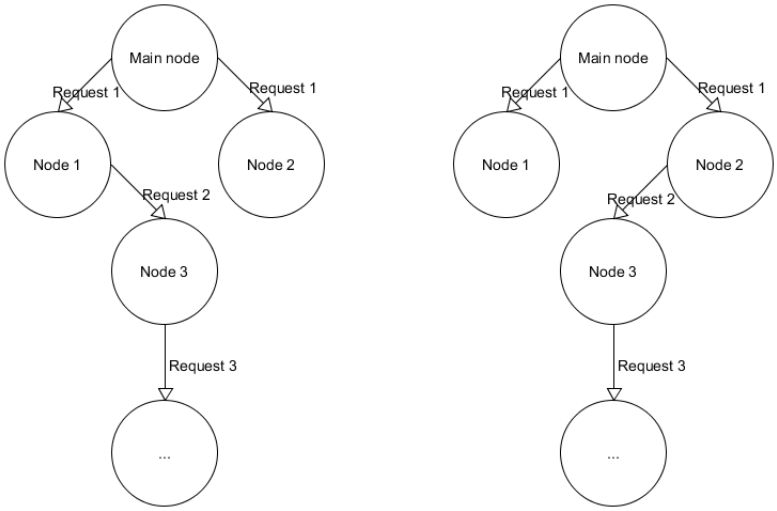

Figure 8.3: The tree structure can vary depending on the node that gets a reuqest to a subnode first.

The tree created, in each wave of request broadcasts, can vary in its structure, depending on interference, removal of nodes, or faulty nodes. Figure 8.3 shows an example of how the tree can vary: first the main node will broadcast *request 1*, which is received by both *node 1* and *node 2*. A sensor node in the system will always handle its own sensor data before relaying data from other nodes. If the main node receives sensor data from *node 1* first, the resulting tree will match the tree seen to the left in figure 8.3. If *node 2* connects to the main node first, the tree will match the tree seen on the right. This will effectively solve the requirement of responding appropriately to a disconnecting node, as well as the inclusion of new nodes in the network, since a new tree is constructed with each new requests from the main node.

In other words, a tree instantiated by the controlled flooding protocol can be

used to construct a tree that spans over a single path from each node to the main node. The receiver of a request signal can use the sender as the parent when data is to be returned. The subnode will then repeat the request signal, and be the parent of nodes hearing it. Any node receiving a request signal will then be able to respond to its parent, causing all nodes to have a single identifier for the routing towards the main node.

Observe that nodes close to the main node are prone to receive a large amount of data since many, if not every, data packet must go though that node. Looking at figure 8.4 it is shown that *node 2* will handle every data packet on the path to the main node. Such low level nodes might be overloaded with data packets, causing a bottleneck if not thoughtfully spread out. The risk of collisions rise if several higher level nodes are trying to send data packets back to *node 2*, but *node 2* has a limited packet throughput. Another scenario is that since *node 2* is relaying data more frequent than other nodes in the system, it consumes more energy and could run out of power before subnodes. This could result in the loss of an entire branch.

8.3.2 Sequence

In this subsection the sequence of how the main and sensor nodes behaves after receiving a packet will be described. The general sequence of the protocol is determined as the flowchart seen in appendix F.

The following is the operation structure for each individual node in the system.

Sensor nodes

1. If the node have not yet been paired up with the main node.

   Send a pair request to the main node.

   Await acknowledgement from the main node and remember the assigned ID.

2. Receive a data request and remember the addresser as parent.
3. Acquire data from sensor and send it to parent.

   Await acknowledgment from parent.

   Resend data if no acknowledgement is received.

4. Generate and broadcast request.
5. Await data from subnodes.

   Respond to subnodes with acknowledgement.

Received data is relayed to parent.

6. If no response or acknowledgment within timeout limit is received, or a clear signal have been received: clear all data except its own ID, relay the clear signal, and await new data request.

Main node

1. When receiving pair request, assign and save ID for the requesting node and respond with an acknowledgement containing the ID.
2. When the greenkeeper requests data in the interface, broadcast a data request.
3. When receiving data from a subnode, save the data and respond with an acknowledgement to the subnode.
4. When data have been received from all registered nodes, broadcast a clear signal and show the data for the greenkeeper.
5. If the timeout limit is reached without being finished, the data is sent and a clear signal is broadcasted.

8.3.3 Interference handling

When utilizing wireless communication, interference can be a problem, as explained in chapter 7.2. This is also the case when using controlled flooding, but the protocol can potentially help reduce problems with interference, for instance by validating the transferred data and acting accordingly.

Interference might cause a change in a packet, which will make the received data unusable. Figure 8.4 shows an example of a possible interference scenario, where nodes 5, 6, and 7 could send packets at the same time, causing node 3 to receive scrambled data.

To counteract this problem, a checksum is used for verifying packets. This checksum is transmitted with the data and recipient. When a node receives a packet, the checksum is calculated, hence verifying the contents of the packet. How the checksum is calculated is explained in section 7.2.2.

To ensure that a packet eventually is received, the nodes will wait a random, but progressively longer, interval, between attempting to send data. This increases the chances for the transmitted data to reach the receiver without interference. Exponential backoff is further explained in chapter 7.1.

It should be noted that the monitored data is not critical. If a packet is lost, it is negligible. If data is not received from one or multiple sensor nodes, a new request can be sent or the node can be repaired or replaced.

8.3.4 Summary

The protocol uses flooding to distribute the data request through the network. When data is requested by the user, the main node sends a packet containing a data request. The nodes that receive this packet directly from the main node is the first level of nodes in the tree, as seen in figure 8.4. These nodes then read the moisture values from their sensors and transmit these back to the main node.

The first level nodes then request data from the nodes within their range. The nodes receiving this request firstly verifies that it is a new request, then registers the node that requested data as its parent. These nodes are second level nodes. The second level nodes read from the sensors, and sends data back to their parents. The parents then relay the data to the main node. This procedure continues until all reachable sensors have relayed data back to the main node. An illustration showing an example of a tree is seen on figure 8.4.

For each new data request from the main node the tree structure and node levels are assessed again as they might have changed, due to new or disconnected nodes.

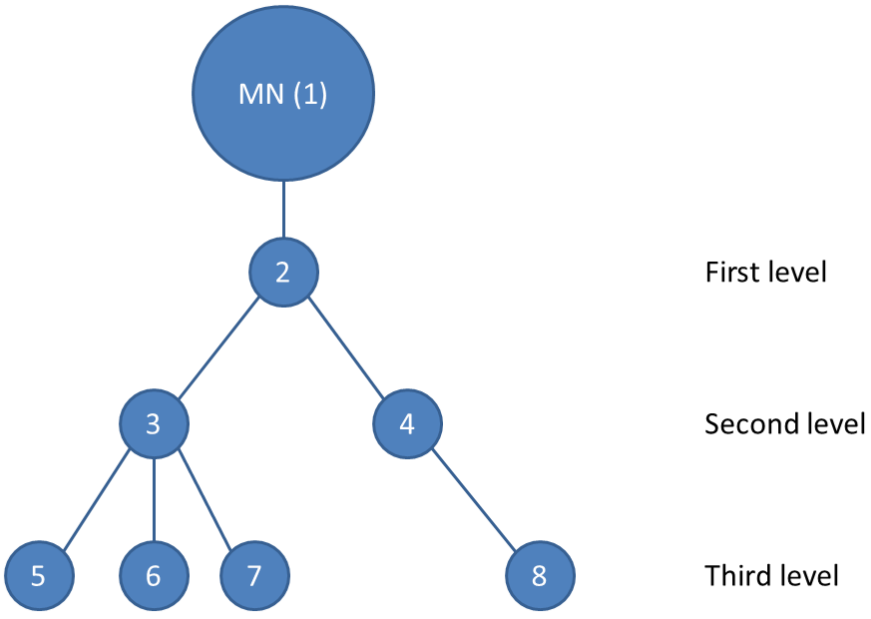

Figure 8.4: Example of a tree.

8.4 Packet

This section covers the descriptions and decisions regarding the packets sent wirelessly in the network.

8.4.1 Packet types

To be able to discern and utilize received data, it is decided to mark the packet accordingly, indicating the packet content. The types are used to inform the receiver what type of data the packet contains, so that the receiving node can decide how to handle the received packet.

There are multiple types of data used in the protocol. As the solution is modular, the protocol is supposed to support other modules; also modules only capable of smaller packet sizes. Though, a packet size of 16 bytes is required to contain the necessary data from the nodes.

Below is a description of the different packet types:

Data
The *data* packet fulfills the main requirement of transmitting the sensor readings to the main node.

Data request
The *data request* packet is the packet notifying the receiving nodes of a data demand. The packet contains the transmitting the ID of the node, so that sensor nodes receiving the packet knows the sender, thus enabling the receiving nodes to set a parent ID.

A lifespan variable is also contained in this packet, used for determining when to stop relaying requests. When the main node requests data, the contents of lifespan is the number of known nodes in the network. The lifespan variable gets decremented by one with every relay. This prevents potential request loops.

Data acknowledgement
A *data acknowledgment* packet is used to confirm that the sent *data* packet has been received correctly. The parent node is responsible for sending a *data acknowledgment* to the addresser of the received packet.

Pair request
A *pair request* packet is sent by an unregistered sensor node and handled directly by the main node, which means it must be within range of the main node. The purpose is to assign an ID to the sensor node, so that every sensor node is registered in the system. The ID's enables the main node to confirm that data is collected from all sensor nodes for the current data gathering, and to identify the origin of the collected data. The ID also provides the capability of a parent-child hierarchy.

Pair request acknowledgement
The *pair request acknowledgement* packet is sent by the main node and used to confirm that the *pair request* has been received and accepted. A *pair request acknowledgement* packet also contain the ID that have been assigned to the specific node. When the two nodes, main and sensor node, are paired, the sensor node can then be deployed on the golf course.

Clear signal
The *clear signal* is used to reset every node in the network, which will happen once the main node have received data from all registered nodes in the system. The *clear signal* packet lets every node in the network know that it should forget any previous information stored, except its own ID. This is done to add flexibility to the system. When *data requests* spread through the network, it forms a tree based on how the nodes are placed. By resetting with the *clear signal*, a new tree could be created each time a new wave of *data requests* goes through the network, which in turn means that nodes can be moved around the golf course between waves of *data requests*.

Error
Because packets are not always transferred correctly, the receiver node will verify the integrity of a packet whenever a packet is received. Section 8.3.3 describes how this is done. If the checksums do not match, the packet type will be set to *error* and discarded.

8.4.2 Addresses

The packet contains the addresser, addressee and the origin address which can be seen in figure 8.5. Due to the potential number of sensor nodes in the network, atleast 2 bytes is required for each address.

Addresser
When a node receives a packet, for instance a packet of the *data request* type, the node needs to respond with its sensor data. The node could begin broadcasting its sensor data openly as response, however this could potentially be received by other nodes in range, whereas the sensor data might be irrelevant. This is why a packet contain the ID of the sender, which is the addresser. The addresser address gives the node knowledge of which node requested data and can therefore address its data correctly. Other nodes in range might receive the data as well, but they will ignore the packet since it was not addressed for them.

The addresser field of a packet will change during the packets lifetime. This happens when a node forwards a packet from another node. The addresser ID will be replaced by the ID of the node relaying the data.

Addressee
When a node wants to send a packet with its sensor data, a data type packet in

this case, it need to specify who this data is meant for, which is the node that requested the data. The ID of the node that requested the data will be marked in the packet as the addressee. This is done to ensure that the packet reaches the intended node. When a node, that is expecting a data packet, receives such a packet, it compares the addressee of the packet with its own ID. The packet will be ignored by the node if the addressee and the ID of the sensor node do not match.

The addressee field of a packet will change during the packets lifetime. This is the case when the packet is relayed, in the same way that the addresser changes.

Origin
The origin contains the ID of the node where the data of a specific packet originated. This data is used by both the main node and the external nodes to differentiate the data.

The origin field of a packet will not be altered during the lifetime of the packet.

8.4.3 Data

When a data packet is constructed, the sensor data is stored in the "data" blocks of the packet as seen in figure 8.5. There are three blocks available for sensor data storage. The solution will, at this time, only contain an implementation of the moisture sensor, but the aim of the three data fields is to add extendibility by enabling support for a maximum of tree sensors per node, each with its own field in the packet.

8.4.4 Checksum

There is a possibility that a packet is not transferred correctly. The packets checksum is stored in the checksum field of the packet and is used for verification.

8.4.5 Packet allocation

The limitation of the network size is 1000 nodes. This means the addresser, addressee and origin address fields need to support 1000 unique addresses, which requires 9 bits. The allocated size for the addresses are accordingly 2 bytes(16 bits) each to maintain usage of whole bytes. This is done because it is more efficient to manipulate a byte instead of bits [44]. The addresses requires 6 bytes in total.

The readings of the moisture sensor have a range of 1024 values, and the required size allocated for the sensor reading data is 10 bits. The allocated data size in the transfer is therefore 2 bytes.

The sizes of the different data to transfer sums up to 16 bytes and the distribution is displayed in figure figure 8.5 as the data 1, data 2, and data 3 fields.

8.4.6 Verification of a packet

The received packet is verified by generating a checksum of message m plus previously generated checksum r. The new generated checksum of a correctly transferred packet T is 0, because a package T, if correctly transferred, is some quotient Q times the polynomial G, as seen in chapter 7.2.2. If a division has a zero remainder, the packet was transferred correctly.

The nRF24L01 supports data payloads up to 32 bytes. Only half of the space available in a transmission is actually occupied. The packet overview can be seen in figure 8.5, where each block represents 1 byte.

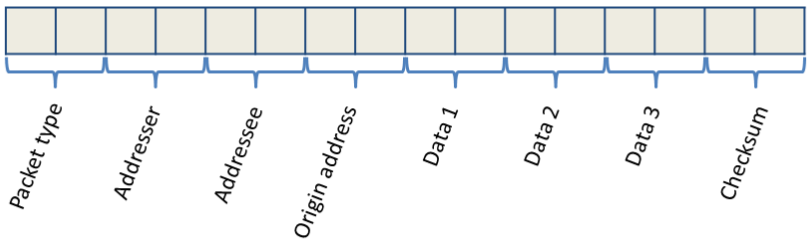

Figure 8.5: Data allocation in packets, where each slot is 1 byte.

8.4.7 Adding and removing nodes

If a new node is to be added to the network, it needs an identifier in the system so that its sensor reading can be recognized. The identifier is permanent as long as it is in the network. The identifier is provided by the main node by pairing the device with the main node.

Should a node somehow disconnect from the network, the nodes connected to this node will receive the packets from another node instead, and from that point on use that as the parent. This is valid if there are any nodes in range of the disconnected children of the nodes.

9. Implementation

This chapter contains documentation about the implementation of the protocol designed in chapter 8.3. The code in this chapter might not be the most complex parts of the code, but it is the most important. The described parts are chosen as they are often used and serves an important role in the solution.

First, the classes used in the implementation are shown and explained along with their methods. Secondly, the important parts of the source code are explained.

9.1 General

The implementation is written in C++. The development of the solution is done using pair programming. Git is used for version control. The implementation is designed according to the requirements of the solution, as well as the decisions made in chapter 8.

Observer that the code in the listings does not necessarily contain the same comments and print statements as in the actual code, to make the code more compact in the report, as these are for debugging.

9.2 Classes

The solution contains two different running applications. The main node and the sensor nodes. There are two sets of code due to the different requirements of the nodes. For example, the main node does not have a sensor, as explained in chapter 8.1.1, so it should not be able to handle a sensor, and therefore does not contain the classes used for sensors.

This section contains the classes contained in each application. UML diagrams showing all fields and methods for both main and sensor nodes can be found in appendix E.

Main node

The main node is a Raspberry Pi, running the Raspbian operating system. The classes in the main node application is seen in figure 9.1. According to the specification, the main node pairs nodes, sends requests, and handles received data.

Sensor nodes

The platforms used for the sensor nodes are Arduino Uno and Mega, although one platform per node. The differences between the two platforms are the pins used for the sensor and radio modules. Besides the pin number, the code is identical on the two platform types.

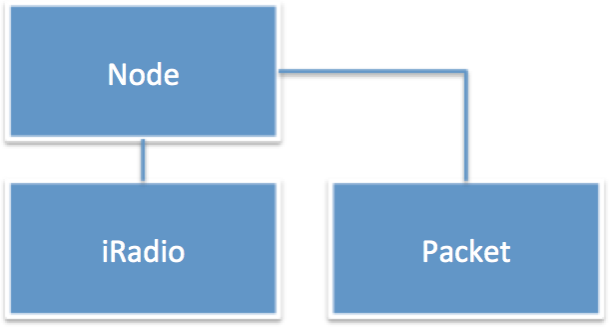

Figure 9.1: Classes used in the main node.

The classes used in the nodes are seen in figure 9.2.

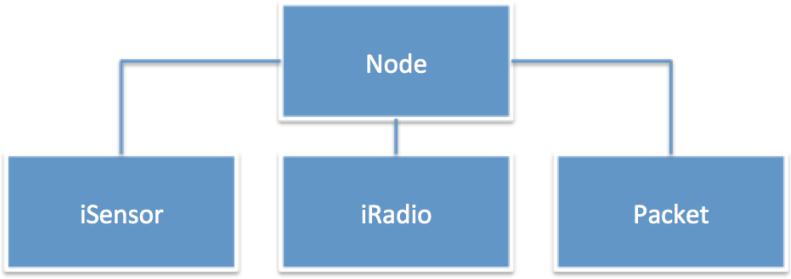

Figure 9.2: Classes used in the sensor nodes.

Class descriptions

This section contains explanations about the classes used in the nodes. As the classes are reused on both main and sensor nodes they are only explained once, even though the code may be slightly different.

Node

The `Node` class is the entry point in the application. On the sensor node, it contains one or more `iSensors`, and a `iRadio` object that are instantiated when the code is executed. The main node contains no sensor, but an `iRadio` object exists.

The `Node` class determines what action to take when a packet is received. If data is received from another sensor node that should be relayed, the `Node` begins sending data until an acknowledgment is received. The methods in the `iRadio` object is used for listening and sending. If data is needed for a packet, for example when a request is received, the `Node` class fetches data from the sensor, and creates the packet that needs to be sent, and makes sure that the data is sent.

iRadio and iSensor

The `iSensor` and `iRadio` interfaces are used to establish a way that all sensors and radio module classes should work. This means that replacing or adding a new sensor or using another radio module is possible, as long as the new component adheres to one of these interfaces. This ensures the rest of the code is able to communicate with these components.

`iSensor` only contains the method `int read()`, that returns the value of the sensor. Sensors can contain different methods internally to verify or handle data, but will be required to have the `read` method.

`iRadio` contains the methods needed for receiving and sending packets. A special listen method is needed to implement exponential backoff as the listen function blocks execution and a way of breaking this blockage is necessary to not lock the code.

void broadcast(char *packet) sends a packet

char *beginListening() begins listening

char *listenFor(int ms) begins listening, but stops listening after `ms` milliseconds

MoistureSensor

The `MoistureSensor` class handles reading from the moisture sensor. This class inherits from `iSensor` and implements the `int read()` method, as required by the interface. Calling `read` returns a value ranging from 0 to 1023.

NRF24Radio

The `NRF24Radio` class handles the radiocommunication. This class inherits from the `iRadio` interface and implements the required methods. The `NRF24Radio` class returns a `char` pointer when a packet is received, using one of the listening methods explained in the `iRadio` section. It is also able to send packets given from other classes, using the `broadcast` method.

Packet

The class `Packet` has the ability to parse a packet from a `char` array to a `Packet` object, along with the properties required to further handle the packet. This includes the type, and possibly sensor values, sender and receiver values or an identifier from the main node.

Instances of this class is created and used in the `Node` class. The objects are created when the radio returns the `char` array from the listen methods.

This class contains the public methods:

Packet(char *input) Create a packet from a `char` array. Used when radio receives a packet.

Packet(PacketType packetTypeInput,
uint16_t addresserInput,
uint16_t addresseeInput,
uint16_t originInput,
uint16_t value1Input,
uint16_t value2Input,
uint16_t value3Input)

Creates a new packet with data as noted in parameters.

char *encode() Returns a char array representation of the packet. Used for broadcasting from the radio.

9.3 Code

This section contains explanations for some of the important parts of the code regarding functionality. First, the `Packet` implementation is explained, with code showing how values are handled and how CRC is implemented. Secondly, the `Node` class is explained along with sensor nodes, and finally the main node and its interface is explained.

9.3.1 Packet

The `Packet` class contains the data that makes a packet. It can be instantiated with a string which is used when receiving a packet in string format from another node using the radio. It can also be instantiated with every part of the packet as a parameter, which is used when a node constructs a new packet to be transmitted. The constructors can be seen in listing 9.1.

```
1 Packet::Packet(PacketType packetTypeInput,
2     uint16_t addresserInput,
3     uint16_t addresseeInput,
4     uint16_t originInput,
5     uint16_t value1Input,
6     uint16_t value2Input,
7     uint16_t value3Input)
8
9 Packet::Packet(char *input)
```

Listing 9.1: Packet constructors

Objects of this class is passed around in `Node`, where it is used to determine actions based on the type, or being relayed by changing `addressee` and `addresser`.

```
1 PacketType packetType; // The type of the packet
2 uint16_t addresser; // The sender of the packet (node ID)
3 uint16_t addressee; // The receiver of the packet (node ID)
4 uint16_t origin; // The original sender of the packet
5 uint16_t value1; // Value 1 (Used for sensor value and lifespan)
6 uint16_t value2; // Value 2
7 uint16_t value3; // Value 3
8 uint16_t checksum; // Checksum value
```

Listing 9.2: Packet variables

Name	packetType	addresser	addressee	origin
Datatype	Packettype	uint16_t	uint16_t	uint16_t
Memory	16 bits	16 bits	16 bits	16 bits

Table 9.1: Overview of first half of a packet.

The datatypes of the members of `Packet` are all `uint16_t`. This is to ensure that the size of the class does not vary on different architectures. The `uint16_t` is always of size 2 bytes. The same size as two characters. Together, all members in the class gives a total size of 16 bytes. The final architecture of a packet can be seen on table 9.1 and table 9.2.

The values shown in listing 9.2 cover the components of a packet; the `packetType`, the `addresser`, the `addressee`, the `origin` of the data, `data` values, and a `checksum`. Every packet has this format, and all values must be filled to ensure uniform size. With packets that might not require all these values, they will simply be 0. For example when sending a request; `addressee`, `value2` and `value3` will be 0.

`PacketType` is defined as shown in listing 9.3.

The `encode` function allocates the size necessary for a `Packet`, but as a `char*` and uses `memcpy` to set the contents of the array to the contents of the current `Packet` and then returning the array. The `char` array is a format of the packet object, that can be directly transmitted by the radio module, without further operations to the array, and then decoded on the receiving node.

The decode function, as shown in listing 9.5, copies the memory of the `char* input` into the memory location where the current `Packet` object is located. This means that the memory contents of the packet is set to the memory con-

Name	value1	value2	value3	checksum
Datatype	uint16_t	uint16_t	uint16_t	uint16_t
Memory	16 bits	16 bits	16 bits	16 bits

Table 9.2: Overview of second half of a packet.

```
enum PacketType : uint16_t {
    Error,
    DataAcknowledgement,
    DataRequest,
    Data,
    PairRequest,
    PairRequestAcknowledgement,
    ClearSignal
};
```

Listing 9.3: Packet types

```
char *Packet::encode()
{
    char *returnstring;
    returnstring = (char*)malloc(sizeof(Packet));

    memcpy(returnstring, this, sizeof(Packet));

    return returnstring;
}
```

Listing 9.4: Packet encode function.

```
void Packet::decode(char *input)
{
    memcpy(this, input, sizeof(Packet));
}
```

Listing 9.5: Decode function.

tent of `input`. This is possible as every part of the packet has a known, predetermined size. `uint16_t` always has the same size, not depending on the architecture running the code.

Checksum

`Packet` objects contains a checksum, used to verify the content's integrity. The checksum uses two functions: `crcInit()` and `crcCompute(unsigned char* message, unsigned int nBytes)`. The checksum implementation is based on the description in `Programming Embedded Systems in C and C++` [45].

The CRC functions share the following data, and this data should be implemented so that it is accessible by the functions without instantiating them multiple times to reduce memory and time consumption.

POLYNOMIAL the generator polynomial by which the message is divided.

INITIAL_REMAINDER the remainder initially appended onto the message before calculating the checksum.

FINAL_XOR_VALUE the value by which the transfer message is inverted.

WIDTH the width of the checksum, which in CRC-16 is 16 bits.

TOPBIT is the most significant bit set to 1.

The `crcInit()` is executed once initially to instantiate a table of remainders to expedite the computation of the checksum. The computation is accelerated because the remainder can be calculated byte-wise instead of bit-wise, by the cost of 256 bytes in the memory.

```
1  void Packet::crcInit()
2  {
3      unsigned short remainder; // 2 byte remainder (according to CRC16/CCITT ↩
           standard)
4      unsigned short dividend;
5      int bit; // bit counter
6
7      for(dividend = 0; dividend < 256; dividend++) //foreach value of 2 bytes/8 bits
8      {
9          remainder = dividend << (WIDTH - 8);//
10
11         for(bit = 0; bit < 8; bit++)
12         {
13             if(remainder & TOPBIT) // MSB = 1 => divide by POLYNOMIAL
14             {
15                 remainder = (remainder << 1) ^ POLYNOMIAL; //scooch and divide
16             }
17             else
18             {
19             remainder = remainder << 1;//scooch and do nothing (MSB = 0, move along↩
                   )
20           }
21         }
```

```
22          Packet::crcTable[dividend] = remainder;//save current crc value in crcTable
23      }
24  }
```

Listing 9.6: Initializes CTC table.

The function iterates over all binary combinations of the 16 bits for the dividend. Within this iteration each remainder is initially instantiated by shifting the dividend by 8 bits, as the purpose is to generate 1 byte values. For each of the 8 bits, if the remainders most significant bit is 1, the remainder is divided by the `POLYNOMIAL` and left shifted one bit. If the most significant bit is 0, the remainder is only left shifted one bit. When all 8 bit computations are finished, the remainder is saved to the `crcTable` in the current `dividend`'s position.

However, the main CRC function is the `getChecksum(unsigned char *message, unsigned int nBytes)`. It utilizes the `crcTable` and computes and returns the checksum of the transfer message based on the arguments `unsigned char *message` and its size, `unsigned int nBytes`.

```
1   uint16_t Packet::getChecksum(unsigned char *message, unsigned int nBytes)
2   {
3       unsigned int offset;
4       unsigned char byte;
5       uint16_t remainder = INITIAL_REMAINDER;
6
7       for (offset = 0; offset < nBytes; offset++)
8       {
9           byte = (remainder >> (WIDTH - 8)) ^ message[offset];
10          remainder = Node::crcTable[byte] ^ (remainder << 8);
11      }
12      uint16_t result = remainder ^ FINAL_XOR_VALUE;
13
14      char *toBeSwapped = (char*)malloc(sizeof(char)*2);
15      memcpy(toBeSwapped, (char*)&result, sizeof(char)*2);
16      char temp = toBeSwapped[1];
17      toBeSwapped[1] = toBeSwapped[0];
18      toBeSwapped[0] = temp;
19
20      memcpy((char*)&result, toBeSwapped, sizeof(char) * 2);
21      free(toBeSwapped);
22      return result;
23  }
```

Listing 9.7: Determines and returns the checksum.

`getChecksum` uses an `int offset` to keep track of the iterative progress through the message, a `char byte` to contain a byte of the `message` and a `uint16_t remainder` instantiated by the `INITIAL_REMAINDER` value.

The function iterates over each byte in the `message`, according to the algorithm described in section 7.2.2. The current `byte` is the remainder right shifted by 8 then XOR'ed with the offset byte of the `message`. The `remainder` is then calcu-

lated by XOR'ing the value already stored in `crcTable` at the current byte, with the remainder left shifted by 8.

9.3.2 Node

The `Node` class is used as a static class, which means that no instances of it is made, whereas instead all methods and variables are bound to the class itself. This class determines the action to take when a packet is received, which takes place in the function `void handlePacket(Packet packet)`, with a `switch` statement deciding with the packet type as argument.

`Node` also makes sure that packets are transferred correctly. This means sending packets until an acknowledgment or other confirmation is received. To achieve this, exponential backoff is used, as explained in chapter 7.1.

The implementation of the `Node` classes are different on the main node and sensor nodes, which is explained in the following two sections.

9.3.3 Sensor nodes

The passive state of the sensor node is listening for packets. When a packet is received, it is handled in `void handlePacket(Packet packet)`.

Listing 9.8 contains the main switch determining what actions to take when a packet is received.

```
1  void Node::handlePacket(Packet packet)
2  {
3      switch (packet.packetType)
4      {
5          case DataRequest: // Received request
6          {
7              if (parentID == -1 && packet.value1 > 0 )
8              {
9                  handleDataRequest(packet);
10             }
11         }
12         break;
13         case Data: // Received data to relay
14         {
15             if (parentID != -1 && packet.addressee == Node::nodeID)
16             {
17                 handleData(packet);
18             }
19         }
20         break;
21         case PairRequestAcknowledgement: // Received ID
22         {
23             handlePairRequestAcknowledgement(packet);
24         }
25         break;
```

```
26          case ClearSignal: // Clear parent and stop sending!
27          {
28              if (parentID != -1)
29              {
30                  handleClearSignal(packet);
31              }
32          }
33          break;
34          default: // Handle everything else (Timeout is handled here)
35          {
36              handleDefault();
37          }
38          break;
39      }
40  }
```

Listing 9.8: The packet handling method.

The cases first determines whether to act on the packet, and then calls the appropriate method. For example when a request is received, the action is only taken if the node has no parent ID, and the packet lifespan is higher than 0. Acknowledgments are not handled in this method, though. It is instead handled in the method that sends data to the node parent, as this is the only place where acknowledgments are relevant, in the method `void beginBroadcasting(Packet packet)`, which handles both relaying and sending data.

Exponential backoff

In the solution, a response is often wanted when sending a packet, and exponential backoff is implemented to make sure that the packet is eventually received. This is used when sending data and waiting for an acknowledgement, or listening for a clear signal.

The code in listing 9.9 shows how a delay is calculated based on the current attempt. The upper bound is calculated by doing $1 \cdot 2^{(attemptNumber-1)}$, and the delay is chosen by generating a random number between 1 and the upper bound.

```
1  unsigned long Node::nextExponentialBackoff(int attemptNumber)
2  {
3      attemptNumber = (attemptNumber <= 32) ? attemptNumber : 32;
4
5      unsigned long potentiallyBiggest = ((unsigned long)1 << (attemptNumber - 1));
6      unsigned long delay = random(1, potentiallyBiggest);
7      delay = (delay < 5) ? 5 : delay;
8
9      return delay;
10 }
```

Listing 9.9: Exponential backoff on the sensor nodes. In Node.cpp.

The variable `attemptNumber` is capped at 32, as there is 32 bits in the datatype `long` on the Arduino, and bitshifting out of the bound, causing overflow, could

cause unwanted results.

Before the delay is returned, the value is set to five if it is lower than five. This is done as 1ms is not enough time for the devices to send, decode, handle, encode and sending a response. Tests showed that acknowledgements for packets being sent with a delay of 1ms were never received, so the packet had to be re-sent. Therefore 5ms was chosen as the lower bound.

9.3.4 Main node

When a packet is received, the values are saved, until all nodes have been accounted for, or a timeout is reached. If a timeout is reached, the data received from nodes will be saved along with a -1 value for the nodes that have not sent any sensor values.

Pairing sensor nodes is done on the main node by generating a unique ID and transmitting it to the sensor node requesting the ID. If no pair requests are received within ten seconds, a new ID will be generated. A file on the main node is used to store IDs and names for the sensor nodes in the network.

When a data request is completed, the data is saved to a file, with the name set as the current date and time. Another file is saved, which is used to determine if a request is done in the web interface, as explained in chapter 9.3.5. A request is started when a signal of the type `SIGUSR1` is received. This, asynchronously, sets a flag, `signalReceived` in `Node` to 1. Each iteration of the main loop will check `signalReceived`, and if set to 1, the requesting will be performed, and the flag set back to 0.

9.3.5 User interface

Users of the system have to be able to start requests and to see data from the sensor nodes when the request is complete. To make this possible, a user interface was developed, that allows users to start requests and see data from this and previous requests.

The interface runs as its own program, as a CGI (Common Gateway Interface) program running on the Raspberry Pi device, using Apache. The executable is located in `/usr/lib/cgi-bin/` and is called `WASP`. The program is written in C, and the complete code can be found in appendix C.

When this program is executed, it generates HTML. Because it is executed as a CGI program using Apache, it is possible to connect to this program using a browser. When connected to the 'website', the user is able to make requests or show previous results. When a link is clicked, a string is appended onto the cur-

rent URL, for example `?request`. The string after the '?' is passed as an argument to the script.

On startup, the program attempts to fetch the process ID of the main node program, located in `/var/run/` if the program is running. This PID is used when the request button is presed by the user. The PID is necessary to call the procedure `kill`, that is used to signal the main node software that it should start a request. The signal is handled as explained in chapter 9.3.4. Kill is a UNIX function that sends a signal to one or many processes. In the solution, the signal `SIGUSR1` is used.

When `kill` is issued, the program redirects to a waiting page. On this page, an AJAX request is performed every second to ping for a file in `/var/www/html/`. This ping will fail until the file `ping.txt` exists at this location. When the request is finished in the main node software, this file is created and the contents is set to the result name. When the file is written and closed by the main node software, the ping will succeed and the contents will be available. The browser can now be redirected to the result from the waiting screen. The ping file is then deleted as it is no longer needed.

The full sequence is shown in figure 9.3. The diagram starts at the webinterface, and the sequence follows the numbers presented on the diagram.

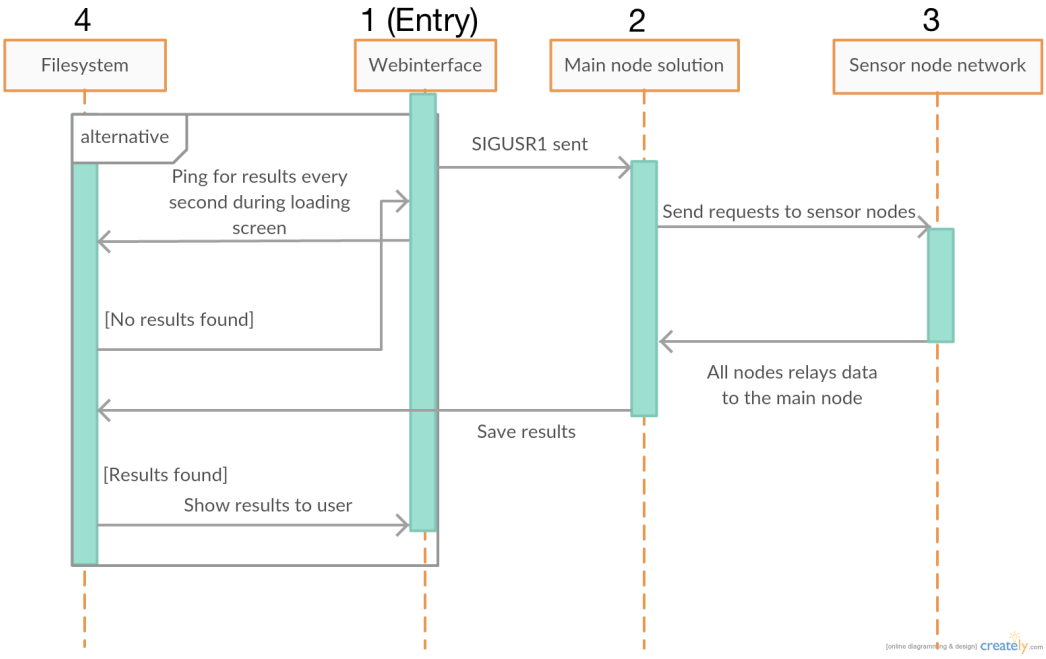

Figure 9.3: Full sequence from webinterface to results[46].

10. Test

This chapter contains documentation about the performed tests. The testing is executed for both the individual nodes and on the entire system, to reveal problems or confirm performance of the system.

The first part of the chapter contains testing done on the nRF24L01 transceiver module used in the nodes. The second part contains testing of individual nodes, including sensors and radio modules. This leads to testing of the entire system, with multiple sensor nodes used, along with the main node.

10.1 nRF24L01

The following section documents the tests performed on the nRF24L01 radio module. These tests were performed to determine different properties of the radio module, such as range and reliability.

10.1.1 Range

A simple test was performed to assess the range of the radio module, to ensure they are usable in a practical application.

The testing was done by having two units set as transmitter and receiver units respectively, having the transmitter repeatedly sending a single signal while increasing the distance between the units. When the receiver no longer received the signal, the distance was measured.

This test showed a range of about 160 meters without obstacles, although the number of successfully transmitted packets declined the further the distance. Observe that at 160 meters the packet loss was around 95%.

10.1.2 Bit Error Rate Test

When transmitting data from one unit to another, it is important that the data is correct. To get an idea of how much error is introduced during the communication, a bit error test was performed, hereby referenced to as BERT.

The basic concept of BERT is to send a known data stream from one unit to another, and check how much it differs from the expected data stream.

The duration of a BERT should vary depending on the results. To get an exact result it would take an infinite amount of time, which is not feasible. But due to the nature of randomness, if only a small amount of errors are met, then it could just be a lucky test, so the testing length would have to be sufficiently long for getting a general idea of the average error rate. Because of this, multiple BERT runs were executed.

Executing BERT

The test was executed by sending packets from an Arduino to a Raspberry Pi, using the nRF24L01 modules.

The code executed on the devices for the test can be found in appendix B. The sender code transmits some deterministic value, and the receiver checks whether the received packet is the same as the one expected. If the two packets match, the 'successes' value is incremented. If not, the 'errors' value is incremented.

The BERT was executed four times. The first three times with 1.000.000 packets transmitted, and the fourth time with 10.000.000 packets. For every 1.000 packet received, the number of errors and successes were printed for those 1.000 packets. The values were then imported into a spreadsheet for statistics. The complete spreadsheet with results and graphs can be found on the attached DVD.

Result of BERT

The results of the BERT showed a packet failure of under one percent. Some spikes were present in the test results, with the highest being at 797 missed packets, which occured in the first test, as seen in figure 10.1.

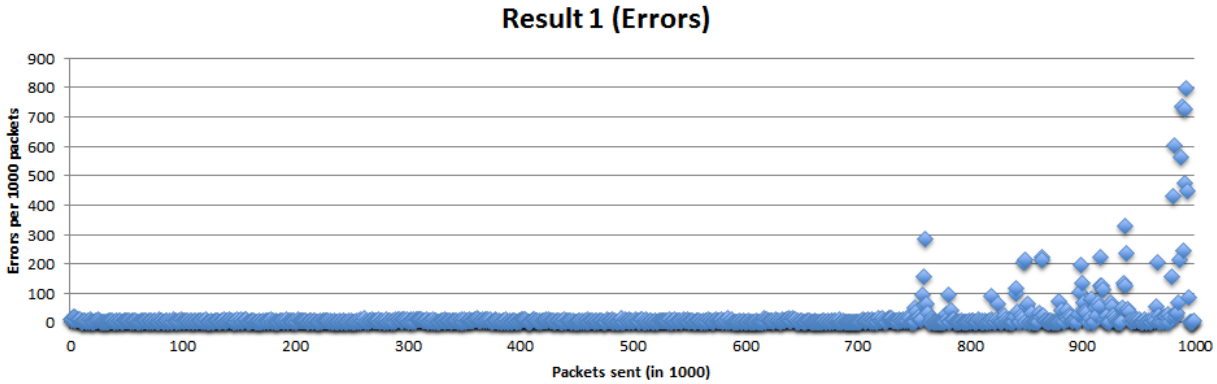

Figure 10.1: Graph representing the first BERT run.

The three last tests generally had a lower number of missed packets. Test number two had a peak of 304 lost packets, as seen in figure 10.2, and test three had a peak of 340 lost packets, figure 10.2. Test four had a peak of 163 lost packets, seen in figure 10.3.

As seen in the first and second test in figure 10.2, the high number of errors occurred in clusters. This could be attributed to the time of the testing, where nearby students might have been testing wireless modules, or some other source of interference occurred.

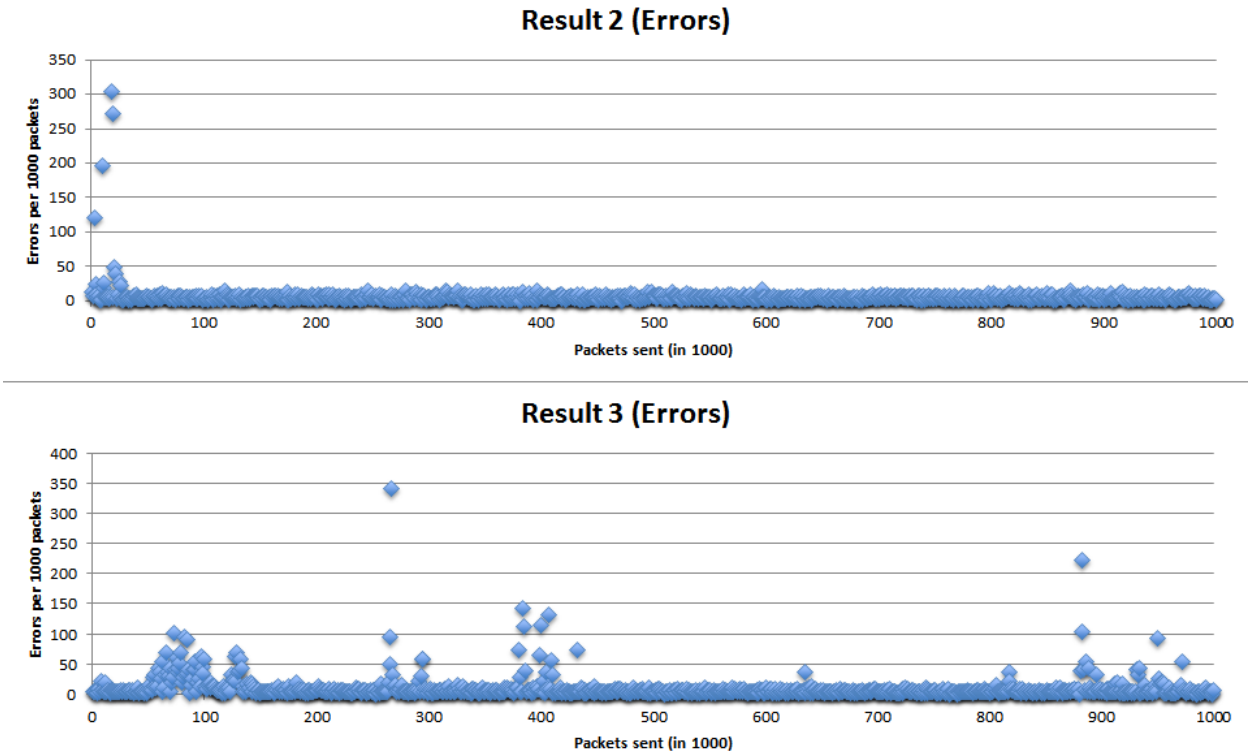

Figure 10.2: Graph representing the second and third BERT run.

The tests were high in success, and low on failures. In the first run, the average failed packets per 1.000 was at 16.5. The second run had an average of 6.4 failed per 1.000. Third run had 9.3 failed per 1.000 and the fourth run had 7.2 failed per 1.000. The results are shown in table 10.1.2.

Run #	Average packets per thousand failed
1	16.5
2	6.4
3	9.3
4	7.2
Avg.	9.85

Table 10.1: BERT results.

The fourth run was performed using ten million packets, and the graph provides better insight on the randomness of errors. A few peaks appeared, but usually smaller peaks between 40 and 60 packets lost per 1.000.

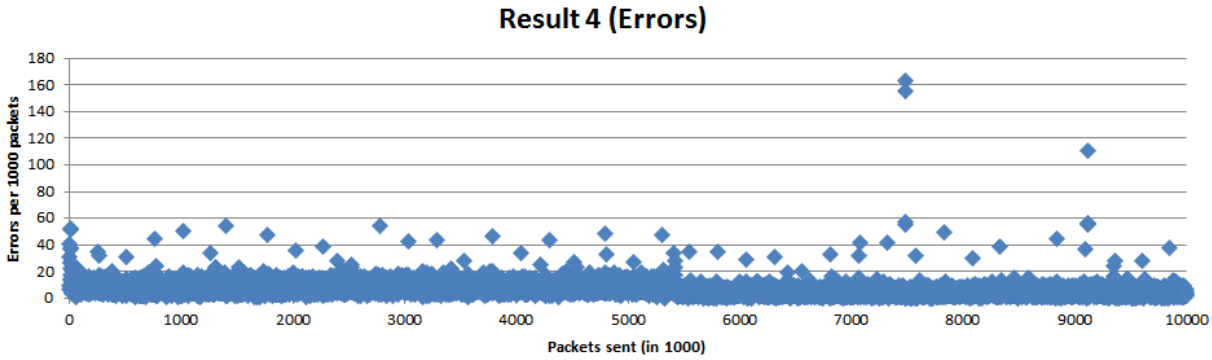

Figure 10.3: Graph representing the fourth BERT run.

Generally, the error count is low and even regarding the peaks, and the module seems stable. On a golf course, there will not be the same level of interference as in the group room, surrounded by wireless networks, phones, bluetooth devices and other sources of interference.

10.2 Code

This section contains tests performed on parts of the code. Most parts of the code are not straight forward to test, as these rely on a signal from a radio module or sensor, but the 'static' parts of the code that can be tested are explained in this chapter. These parts are tested to make sure they function as intended.

The `Packet` class has been tested during development to ensure correct behavior in the class. As some of the functions in the class performs low-level memory operations, it is important to confirm that everything performs as expected. As the `Packet` class is the same on the main node and sensor nodes, only one test is needed.

Before testing this class, the methods `bool checksumMatches()` and `bool compare(Packet)` were defined, to aid in testing the class.

Encoding and decoding

The encoding and decoding of the packet classes were tested by constructing a set of packets, and then encoding and decoding these packets. The code used for this is seen on Listing 10.1.

```
1 // Create and encode packet
2 Packet packet1(DataRequest, 1, 2, 3, 4, 5, 6);
3 char *packet1Encoded = packet1.encode();
```

```
4
5  // Create packet from encoding
6  Packet packet2(packet1Encoded);
7
8  // Compare!
9  if(packet1.compare(packet2))
10 {
11     std::cout << "Packets are the same." << std::endl;
12 }
13 else
14 {
15     std::cout << "Packets are NOT the same!" << std::endl;
16 }
```

Listing 10.1: Testing packet encoding/decoding.

The code in Listing 10.1 starts out by encoding and decoding a single `Packet`, to do a simple test. The variable `packet2` is created from the encoding of `packet1`.

The tests showed that packet encoding and decoding works in the solution.

Valid and invalid packets

The solution uses a checksum to validate packets received, as explained in chapter 8.3.3. Testing whether the implementation of checksums work is done by using the code found in Appendix D.

The code initializes a `Packet` object and verifies that the checksum matches the content of the packet. The values in the packet is then changed, one by one, and the checksum is tested again. After each test, the result is printed and the value is set to the initial value. After testing changing each value, the packet is verified again with all the default values.

The tests showed that any value being changed will make the packet invalid. When all values are set to the initial value again, the packet is valid again. This proves that if a packet is changed while transferring data, some part of the packet would not be correct and the packet will be rejected.

10.3 System

This section contains tests performed on the complete system after the code was deemed complete. To test the entire system, multiple nodes have to be setup and ready. The tests explained in this chapter were performed with a main node, and two sensor nodes.

The main node and sensor nodes have different tasks in the system, so the test parameters are different. Below is the tasks that were performed during the testing, for both node types.

Sensor nodes

1. Send pair request if an ID is not found on the EEPROM.
2. Save node ID on EEPROM when an ID is received from main node.
3. When receiving a request, save the sender ID as parent, and send data back to this node. If a parent is already known, ignore this request.
4. When receiving acknowledgement of data from parent node, stop sending data and send requests.
5. When receiving data from a subnode, send acknowledgement to this subnode.
6. After sending acknowledgement, relay data to parent node until acknowledgement is received.
7. When a clear signal is received, clear parent and stop all actions. Relay clear to other nodes in the vicinity.

Main node

1. When receiving a pair request, generate ID and reply to node.
2. When a user requests data in the interface, broadcast data requests.
3. When receiving data from subnode, save data and send acknowledgment to sender.
4. When data is received from all nodes, save data in file and show user results to the user. Send clear when file is saved to stop any action in the network.

The tests above covers the basic functionality of both main node and sensor nodes. They describe what the nodes should do from they startup to a request is finished. Testing these tasks can be done by observing a node that is part of the system, and test whether it does what it is supposed to. This requires multiple tests, as all nodes have to be observed at some point to determine wether the nodes do what they are supposed to do.

Due to the long range of the radio modules, relaying had to be tested using special code to ensure that a direct transmission between the main node and all sensor nodes. This meant that one of the sensor nodes was running code that ignored packets from the main node, to force the node into becoming a subnode of the other sensor node used in the test.

Results

By monitoring sensor nodes during a request, it was possible to see that the nodes reacted in the correct way to requests. First, by sending their own data to the parent node, whose ID was saved when the request arrived, and then stopping when an acknowledgment was received from the parent.

The sensor node then sent out requests, but with itself as the addresser. The other node received the request and then followed the same procedure as the first node. Upon receiving data, the first node relayed this data to the main node.

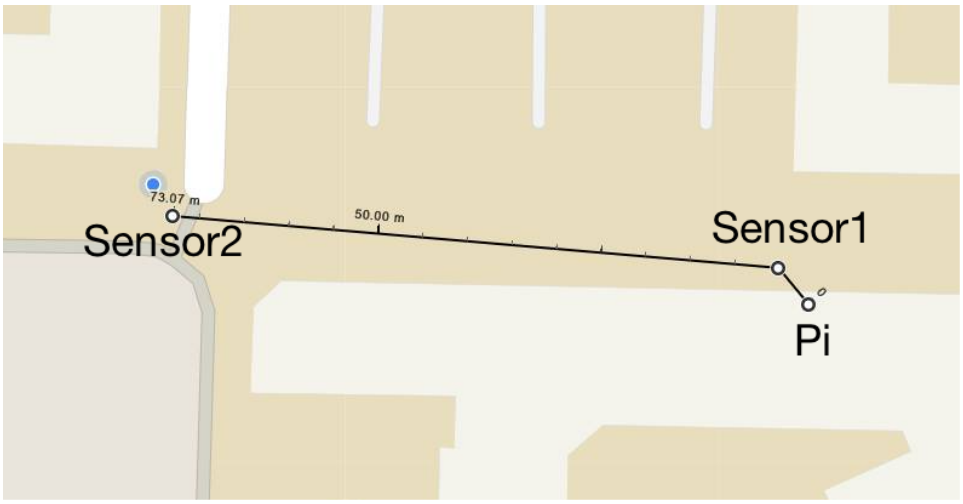

Figure 10.4: System test on map, showing nodes.

The main node now had data from all sensor nodes. This data was saved to a file, and clear requests were sent. The clear requests were then relayed from the first level sensor node to the second one and the request was complete. Figure 10.4 shows the setup of the test. The `Raspberry Pi` was inside the group room with both sensor nodes being outside.

This proved that the sensor nodes and main nodes completed their tasks. Testing showed an error in the implementation of exponential backoff, which made it possible for the delay to go out of bounds on the Arduino platforms. This was corrected after discovery.

11. Summary

In this chapter, the project will be evaluated on different parameters. First, the working process and tools will be evaluated to clarify what methods were good, and bad. The final outcome of the project will be evaluated, and is the report conclusion, which evaluates the problem statement to the outcome. In the final section, ideas for further improvements are listed. Furthermore the system will be compared to other use cases, since the system might solve other problems than measuring soil moisture on a golf course.

11.1 Reflection

The working process has led the project group some powerful tools in the project. The choice of developing a prioritized requirements specification made the group able to utilize the whole project period with 'nice to have' features if the main requirements were to be implemented too early, and having an iterative implementation process.

In the design and implementation phase there were several iterations because subproblems were found several times. These problems had to be described, analysed and handled in the solution before proceeding. Though, this was to be expected due to none of the group members having developed a protocol before.

Pair programming worked out well in the group, and led to greater understanding of the code. Since at least two members in the group were able to explain the thoughts behind the code, information sharing among the whole group could be done in a simpler manner. Furthermore, pair programming led to a higher quality solution because a pair could discuss potential implementations before actually choosing one.

The planning tool, Trello, used in the project did cause problems, as members in the group forgot to update Trello. Therefore it was harder to track the progress and state of all tasks. Still, the group did not run into any conflicts or 'double work' during the project.

The solution fulfills the requirements, but some aspects from the analysis were completely left out. Explicit power saving features were not implemented anywhere, which resulted in a system that might run out of power. Some functions have been implemented with power saving in mind, eg. that the system operates on demand instead of automatically monitoring, or that a node will eventually stop trying to broadcast its data.

For the solution to be finally verified as functioning, it would be required to test the solution on an actual golf course with a realistic number of nodes and some actual users. This is the only way to verify that the system actually works in a real situation.

11.2 Conclusion

The result of the project is a protocol, enabling a network of devices to transmit data wirelessly through the network to a main node. As sensor nodes are able to relay data through the network, the main node does not have to be in range of all sensor nodes to receive their data.

The analysis was based on an interview with a greenkeeper, as the project group had no prior knowledge of the problem domain. A thorough analysis was made and led to the following problem statement: *How can a sensor network and a corresponding protocol be designed for a golf course, so that data can be relayed throughout the network, enabling an endpoint device to receive the information without being within range of all sensors in the network?*

The problem statement then led to a requirement specification. The design phase led to a final design where technologies for the protocol were chosen. The protocol was implemented according to the design and using a customized working process.

According to the tests, a small network of nodes, can fulfill the requirements of reading sensor data, transmitting and relaying data through a network, as well as displaying the measurements in a user interface, thus successfully solving the problem statement.

11.3 Future work

This section contains ideas and requirements for further developing the solution, and even implementing it on a real golf course. Furthermore this system can be used in other cases with a slightly similar problem domain, which will also be described in this section.

11.3.1 Functionality

To make the solution work on an actual golf course it would also be required to replace the radio modules because the small modules used in the project have a very limited range. This would, however, only require a hardware replace and class replace in the software because all transmitting and receiving functionality is done within the Radio class.

For the user to understand and use the collected data, it is needed to revise the presentation of the collected data. Now, the interface presents the raw data received from the node, which can confuse the user. Therefore raw data should be converted to meaningful data such as water percentage.

The system should utilize the received data in a more useful way. For example by constructing graphs or other presentations involving more data. Another aspect could be including weather data and link them to moisture reading. Then, after a certain amount of collected data, the system could predict the amount of watering needed based of weather forecasts. The system could even be expanded to cooperate with an automatic watering system.

The solution could also be further developed to monitor several properties. Since the data packets supports three data values, more data could simply be added to a packet and transmitted to the main node.

Finally the system should be optimized, witb regards to power consumption. Currently the system has not been tested for power consumption. The nodes should include functionality to save power when not operating.

11.3.2 Other use cases

This system is not limited to soil moisture on golf courses alone. The system can, for example, also be used on fields, enabling farmers to monitor the soil moisture. The system could also be used to monitor gardens or even houseplants - everywhere where soil moisture is relevant to check regularly.

By changing the sensors the solution could be used in any problem domain consisting of the need for a multiple spot monitoring system, where rolling out cables would be challenging.

Such a domain could be in the environment, where rock- or snowslides can occur. To avoid a catastrophe, a monitoring solution could warn nearby people, and potentially save lives.

Bibliography

[1] Ian Sommerville. *Software Engineering*. Pearson Education Inc., 2010. ISBN: 9780137035151.

[2] R & A. *The R & A - The Golf Course*. Seen 29/09/2015. URL: http://www.randa.org/en/Playing-Golf/The-Golf-Course.aspx.

[3] R & A. *Managing for healthy Golf Course Grass - Turf Management - R & A Golf Course Management*. Seen 01/10/2015. URL: http://golfcoursemanagement.randa.org/en/Managing-Your-Course/Managing-for-healthy-grass.aspx.

[4] Wikipedia. *Soil*. Seen 15/12/2015, URL: https://en.wikipedia.org/wiki/Soil#Soil_moisture_content.

[5] Rain Bird. *SMRT-Y Soil Moisture Sensor Kit*. Seen 08/10/2015. URL: http://www.rainbird.com/landscape/products/accessories/smrty.htm.

[6] Rain Bird. *SMRT-Y SOIL MOISTURE SENSOR KIT - Manual*. Seen 08/10/2015. URL: https://www.rainbird.com/documents/turf/bro_SMRTY.pdf.

[7] The Toro Company. *Turf Guard Sensors*. Seen 08/10/2015. URL: http://www.toro.com/en-gb/golf/irrigation/sensors/Pages/Model.aspx?pid=turf-guard-sensors.

[8] The Toro Company. *Turf Guard Wireless Soil Monitoring System*. Seen 08/10/2015. Requires Flash. URL: http://www.toro.com/irrigation/golf/turfguard/micro/index.html.

[9] Arduino. *Arduino - ArduinoBoardUno*. Seen 16/09/2015. URL: http://arduino.cc/en/Main/ArduinoBoardUno.

[10] Arduino. *Arduino - Introduction*. Seen 16/09/2015. URL: http://arduino.cc/en/Guide/Introduction.

[11] Arduino. *Arduino - ArduinoBoardMega*. Seen 16/09/2015. URL: https://www.arduino.cc/en/Main/arduinoBoardMega.

[12] Arduino. *Compare board specs*. Seen 12/10/2015. URL: https://www.arduino.cc/en/Products/Compare.

[13] Seen 17/09/2015. URL: https://www.robotics.org.za/image/data/Arduino/Arduino%20Boards/arduino_mega_r3_002_hd.jpg.

[14] Bill Traynor. *Raspbian*. Seen 29 November 2014, URL: http://elinux.org/Raspbian.

[15] Raspberry Pi Foundation. Seen 01/10/2015. URL: https://www.raspberrypi.org/wp-content/uploads/2014/07/rsz_b-.jpg.

[16] .

[17] Smart Prototyping. *Soil Hygrometer Detection Module Soil Moisture Sensor For Arduino*. Seen 29/10/2015. URL: http://smart-prototyping.com/Soil-Hygrometer-Detection-Module-Soil-Moisture-Sensor-For-Arduino.html.

[18] Inc. Bluetooth SIG. *A Look at the Basics of Bluetooth Technology*. Seen 24/09/2015. URL: http://www.bluetooth.com/Pages/Basics.aspx.

[19] SparkFun. *XBee Buying Guide*. Seen 24/09/2015. URL: https://www.sparkfun.com/pages/xbee_guide.

[20] Digi. *ZigBee Wireless Standard*. Seen 24/09/2015. URL: http://www.digi.com/technology/rf-articles/wireless-zigbee.

[21] ITU International Telecommunication Union. *ITU - Frequently asked questions*. URL: http://www.itu.int/net/ITU-R/terrestrial/faq/index.html.

[22] Nordic semiconductor. *nRF24L01 Product Specification*. Seen 13/10/2015. Datasheet can also be found on attached DVD. URL: http://www.nordicsemi.com/eng/nordic/download_resource/8041/1/11165739.

[23] K. Mansfield and J. Antonakos. *Computer Networking for LANS to WANS: Hardware, Software and Security*. Networking (Course Technology, Inc.) Cengage Learning, 2009. ISBN: 9781423903161. URL: https://books.google.no/books?id=VQvhAN9iBuMC.

[24] K. Rosen. *Discrete Mathematics and Its Applications 7th edition:* McGraw-Hill Education, 2012. ISBN: 9780073383095. URL: https://books.google.no/books?id=XcrKi-kzwLIC.

[25] J.M. Kizza. *Guide to Computer Network Security*. Computer Communications and Networks. Springer London, 2015. ISBN: 9781447166542. URL: https://books.google.dk/books?id=d2CYBgAAQBAJ.

[26] Halvor Bothner-By. *protokoll - IT - Store norske leksikon*. Seen 15/10/2015. URL: https://snl.no/protokoll%2FIT.

[27] Dr. Anis Koubas. Slide 6 - seen 14/10/2015. URL: http://www.coins-lab.org/pnu/akoubaa/net331/Lectures/Lecture13-Broadcast-Multicast.pdf.

[28] *Flooding (computer networking)*. Seen 8/10/2015. URL: https://en.wikipedia.org/wiki/Flooding_(computer_networking).

[29] *Reverse Path Forwarding*. Seen 8/10/2015. URL: https://en.wikipedia.org/wiki/Reverse_path_forwarding.

[30] Lou Frenzel. *Fundamentals of Communications Access Technologies: FDMA, TDMA, CDMA, OFDMA, AND SDMA*. Seen 17/09/2015. URL: http://electronicdesign.com/communications/fundamentals-communications-access-technologies-fdma-tdma-cdma-ofdma-and-sdma.

[31] Jim Corenman. *RF Interference*. 10/14/99. URL: http://siriuscyber.net/rfi.htm.

[32] UIUC D. Maltz Microsoft Research D. Johnson Y. HU. *The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4*. February 2007. URL: https://www.ietf.org/rfc/rfc4728.txt.

[33] Samir Das Ian Chakeres Charles Perkins Elizabeth Belding-Royer. *AODV*. Seen 17/09/2015. URL: http://moment.cs.ucsb.edu/AODV/.

[34] Johns Hopkins Dr. Baruch Awerbuch Dr. Amitabh Mishra. *Ad hoc On Demand Distance Vector (AODV) Routing Protocol*. Seen 17/09/2015. URL: http://www.cs.jhu.edu/~cs647/aodv.pdf.

[35] Jean-Philippe Lang. *B.A.T.M.A.N. protocol concept*. Seen 27/09/2015. URL: http://www.open-mesh.org/projects/open-mesh/wiki/BATMANConcept.

[36] Stephane Rousseau Vania Conan. *Route Selection for Capacity Maximization in Multi-Rate TDMA-based Wireless Ad Hoc Networks*. Seen 16/09/2015. URL: http://cnd.iit.cnr.it/rbruno/files/meshtech09.pdf.

[37] Piotr ZWIERZYKOWSKI Piotr OWCZAREK. *ROUTING PROTOCOLS IN WIRELESS MESH NETWORKS - A COMPARISON AND CLASSIFICATION*. 2009. URL: http://www.academia.edu/5756527/ROUTING_PROTOCOLS_IN_WIRELESS_MESH_NETWORKS_-A_COMPARISON_AND_CLASSIFICATION.

[38] Audio-Technica. *Types of Interference*. Seen 15/12/2015, URL: http://www.audio-technica.com/cms/site/6d4b2edb868000db/.

[39] Halvor Bothner-By. *Datapakke*. Seen 15/11/2015, last updated 17/10/2012. URL: https://snl.no/datapakke.

[40] T. Boyles. *CCNA Security Study Guide: Exam 640-553*. Wiley, 2010. ISBN: 9780470636336. URL: https://books.google.dk/books?id=AHzAcvHWbx4C.

[41] A. Elahi. *Network Communications Technology*. Delmar, 2001. ISBN: 9780766813885. URL: https://books.google.dk/books?id=g6Vu7vu20rgC.

[42] H.S. Warren. *Hacker's Delight*. Pearson Education, 2012. ISBN: 9780133085013. URL: https://books.google.dk/books?id=VicPJYM0I5QC.

[43] David J. Wetherall. Andrew S. Tanenbaum. *Computer networks*. 5th edition. Pearson Education, Inc., 2011. ISBN: 9780132126953.

[44] Allen Sherrod. *Data Structures and Algorithms for Game Developers*. 2007. URL: https://books.google.dk/books?id=7bILAAAAQBAJ&pg=PA66&lpg=PA66&dq=use+whole+bits+instead+of&source=bl&ots=7vfwbvguQm&sig=UHRktfG_YyJiCYgkdXqWz60vABM&hl=en&sa=X&ved=0CB4Q6AEwAGoVChMIiNr_x4KayQIVShIsCh0pcwYr#v=onepage&q=use%20whole%20bits%20instead%20of&f=false.

[45] M. Barr. *Programming Embedded Systems in C and C++*. O'Reilly Series. O'Reilly, 1999. ISBN: 9781565923546. URL: https://books.google.no/books?id=a2Q6U0b36rMC.

[46] Cinergix Pty. Ltd. *Online Diagram Software to draw Flowcharts, UML & more Creately*. Seen 07/12/2015. URL: https://creately.com.

Appendix

A. Interview

Two interviews were performed with Kim. The first one to gain an understanding of which problems greenkeepers might have, that could be solved with an embedded system. The second one to verify the usefulness of the idea to monitor properties of the golf course.

Initierende interview

Hvilke ting holder i øje med, i henhold til vedligeholdelse af golfbanen?
Hvordan beplantningen som træer og græs gror.

Er det nogle arbejdsopgaver i kunne se automatiseret?
Automatiserede klippere, hvilket dog allerede er lavet.

Med et middel der kunne gøre, at vi kunne se jordens tilstand med surhed og vandmængde, ville vi lettere kunne tilføre hvad der mangler af gødning og vand.

Ville en automatisering resultere i et besparelse eller anden fordel for jer?
Set fra golfbanens ledelse så ville automatiserede maskiner være gode, da de ville kunne spare greenkeeperen. Dette ville dog ikke være til vores fordel som greenkeepere

Uddybende interview

Hvor vigtigt er det for jer at undersøge jordfugtigheden?
Det er vigtigt på greens i forhold til, om de tørrer ud. Dette sker ikke så ofte, men er stadig relevant. Det kan også være relevant på teesteder.

Hvordan gør i det nu?
Når hullerne flyttes undersøges dette.

Hvor ofte undersøger i det nu?
Mest om sommeren, da der oftest sker udtørring. Jeg planlægger det efter vejret, og her kan produktet være til hjælp for at spare dette arbejde.

Ville det være attraktivt med et system der automatisk undersøger det?
Ja det vil det. Der kan tilmed være forskel på forskellige steder på banen, da det kan regne nogle steder uden at det gør andre.

Kender i allerede til systemer til dette formål? (Her refereres til undersøgelse af jordfugtigheden, som tidligere er nævnt)
RainBird har et lignende system. Disse har også leveret styreprogram til vandingsanlæget til Aalborg Golfklub.

Vil det give mening at sende informationerne trådløst i stedet for gennem kabler?
Ja, så slipper vi for at grave dem over hver gang vi skal reparere noget på banen.

Dette kan give problemer med allerede eksisterende ledninger og vandingsanlæg. Især også med rødder på træer.

Hvor på banen skulle sensorer placeres? (Og hvordan? dybde? jord/sand?)
Mellem 10-15 cm. nede, da der skal være rødder på greenen. Der er ca. 30cm. sand på greenen, som vandet hurtigt ryger igennem. Rødder er ca. 10-15cm. nede, der hvor man gerne vil have det er fugtigt. En green kan være ret stor, så det kan være nødvendigt at placere flere på en green. En green kan sagtens være $500 m^2$.

Teesteder er ikke så vigtigt.

Ser du nogen potentielle problemstillinger ved et sådant system?
Når jorden prikkes kan de rammes og evt. gå i stykker. Kan undgås ved at grave lidt ned. Jeg mener at de prikker ca. 10cm. ned. At finde dem efter de er gravet ned kan hurtigt blive et problem. Batteritid er vigtigt og kan også blive et problem.

Nogle idéer til anvendelse eller forbedringer/udvidelser til systemet?
pH-meter indbygget vil være smart i forhold til at udregne brugen af gødning og kalk for at få den ønskede pH værdi (Græs gror bedst ved pH 5.6, men varierer efter græssort, men som regel under 7) Hvorvidt jorden er porøs eller ikke porøs, da dette skal bruges til at finde ud af om jorden skal prikkes. Jord har godt af at blive prikket, da for tæt jord holder på vandet og dette kan stoppe alt vækst og materiale.

This chapter contains the code used in the Bit Error Rate Test. Run on Arduino and Raspberry Pi devices.

B.1 Sender

The sender code runs on an Arduino Uno. This code transmits the string called 'text' in the code, and waits 2ms before transmitting again.

```
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

//RF24 radio(53,48); // Mega
RF24 radio(7, 8); // Uno
const byte rxAddr[6] = "00001";

void setup()
{
  Serial.begin(9600);
  radio.begin();
  radio.setAutoAck(false);
  radio.setDataRate(RF24_250KBPS);
  radio.setPALevel(RF24_PA_MIN);
  radio.setCRCLength(RF24_CRC_DISABLED);
  radio.setChannel(114);
  radio.openWritingPipe(rxAddr);

  radio.stopListening();
}

void loop()
{
  const char text[] = "10101010101010101010101010101010";
  // "10101010101010101010101010101010";
  // "11111111111111111111111111111111";
  // "01010101010101010101010101010101";
  radio.write(&text, sizeof(text));
  delay(2); // 2 default
}
```

B.2 Receiver

The receiver code is run on the Raspberry Pi. This code receives packets and checks them against the expected packet. If the packet is correct, the correct value is incremented. If not, the error value is incremented. This is then printed to a file and the console.

```
#include <cstdlib>
#include <iostream>
#include <sstream>
```

```
4  #include <string>
5  #include <RF24/RF24.h>
6
7  using namespace std;
8
9  void setupRadio();
10 void receiveLoop(unsigned int goal, unsigned int interval);
11
12 //Setup radio module on raspberry pins.
13 RF24 radio(22,0);
14
15 //Adress for data packages
16 const uint8_t rxAddr[6] = "00001";
17
18 int main(int argc, char** argv)
19 {
20   unsigned int goalK = 10;
21   unsigned int intervalK = 10;
22
23   if (argc >= 2)
24   {
25     goalK = atoi(argv[1]);
26   }
27
28   if (argc >= 3)
29   {
30     intervalK = atoi(argv[2]);
31   }
32
33   setupRadio();
34   receiveLoop(goalK*1000,intervalK*1000);
35
36   return 0;
37 }
38
39 void setupRadio()
40 {
41   radio.begin();
42   radio.setAutoAck(false);
43   radio.setDataRate(RF24_250KBPS);
44   radio.setPALevel(RF24_PA_MIN);
45   radio.setCRCLength(RF24_CRC_DISABLED);
46   radio.setChannel(114);
47   radio.openReadingPipe(0, rxAddr);
48   radio.startListening();
49 }
50
51 void receiveLoop(unsigned int goal, unsigned int interval)
52 {
53   const char emptyString[32] = {0};
54   char receivedString[33] = "00000000000000000000000000000000";
55   char expectedString[33] = "10101010101010101010101010101010";
56
57   unsigned int sessionErrors = 0;
58   unsigned int sessionSuccesses = 0;
59   double sessionErrorRate = 1.0;
60
61   unsigned int totalErrors = 0;
62   unsigned int totalSuccesses = 0;
63   double totalErrorRate = 1.0;
64
65   unsigned int sessionCount = 0;
```

```
66    unsigned int intervalCount = 0;
67
68    FILE *fp;
69    fp = fopen("result.txt","w+");
70
71    while(totalErrors+totalSuccesses < goal)
72    {
73      if (radio.available())
74      {
75        strcpy(receivedString,emptyString);
76        //memcpy(receivedString,emptyString,32);
77        //printf("\nreceivedString before receive: %s",receivedString);
78        radio.read(&receivedString, sizeof(receivedString));
79        //printf("\nreceivedString after receive: %s",receivedString);
80        if(0 == strcmp(receivedString,expectedString))
81        {
82          totalSuccesses++;
83          sessionSuccesses++;
84        }
85        else
86        {
87          totalErrors++;
88          sessionErrors++;
89        }
90        sessionCount++;
91
92        //print stuff
93        if(sessionCount >= interval)
94        {
95          intervalCount++;
96          if(sessionSsuccesses > 0)
97          {
98            sessionErrorRate = (double)sessionErrors / sessionSuccesses;
99            totalErrorRate = (double)totalErrors / totalSuccesses;
100         }
101         printf("Total packets received: %d * %dk:\n\tSession errors: %d,\tSession ↩
                successes: %d,\tSession error-rate: %f%%\n\tTotal errors: %d,\tTotal ↩
                successes: %d,\t\tTotal error-rate: %f%%\n", intervalCount,interval↩
                /1000, sessionErrors, sessionSuccesses, sessionErrorRate*100, ↩
                totalErrors, totalSuccesses, totalErrorRate*100);
102         fprintf(fp, "%d,%d\n",sessionErrors,sessionSuccesses);
103         sessionCount = 0;
104         sessionErrors = 0;
105         sessionSuccesses = 0;
106       }
107     }
108     fflush(stdout);
109   }
110   fclose(fp);
111 }
```

```
1  #define _POSIX_SOURCE
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <signal.h>
6  #include <dirent.h>
7  #include <map>
8
9  void printNewlines(int);
10
11 int main(int argc, char *argv[])
12 {
13   std::map<int, char *> nodeMapping;
14     int sendRequest = 0;
15     char openFileName[50];
16
17     // Argumenter!
18     if(argc == 2)
19     {
20         if(strcmp(argv[1], "request") == 0) // Send request!
21         {
22             remove("/var/www/html/ping.txt");
23             sendRequest = 1;
24         }
25         else if(strcmp(argv[1], "success") == 0) // Request er sendt!
26         {
27             sendRequest = 2;
28         }
29         else // Filename, hopefully
30         {
31             sendRequest = 3;
32             strcpy(openFileName, argv[1]);
33         }
34     }
35
36     // Find solution PID
37     int solutionPID = -1;
38     FILE *pidFile;
39     pidFile = fopen("/var/run/wasp.pid", "r");
40     if(pidFile != NULL)
41     {
42         fscanf(pidFile, "%d", &solutionPID);
43     }
44
45     // Header
46     printf("Content-Type: text/html\n\n");
47
48     // Indhold start
49     printf("<html><head><meta http-equiv='Content-Type' content='text/html; charset↩
           =UTF-8'/><link rel='stylesheet' type='text/css' href='/style.css'><title>↩
           WASP - Administration</title></head><body>");
50
51   // Læs navne fra wasp.conf!
52   FILE *optionsFile;
53     optionsFile = fopen("/home/pi/wasp/wasp.conf", "a+");
54     if(optionsFile != NULL)
55     {
56         int nodeID = 0;
57         char nodeName[100];
58       while(fscanf(optionsFile, "%d*%s", &nodeID, &nodeName) != EOF)
59       {
60         // Haxx.
```

```
61          char *tmp;
62          tmp = (char *)malloc(100*sizeof(char));
63          strcpy(tmp, nodeName);
64
65              nodeMapping[nodeID] = tmp;
66          }
67          fclose(optionsFile);
68      }
69
70      // Header
71      printf("<div id='head'><h2><a href='/cgi-bin/WASP'>WASP Administration</a></h2><p>Wireless Arduino Sensor Protocol</p></div>");
72
73      // Nav
74      printf("<div id='nav'>");
75      if(sendRequest == 0)
76      {
77          printf("<input type='button' value='Hent data' onclick=\"window.location='?request';\">");
78      }
79
80      // PID for solution
81      if(solutionPID != -1)
82      {
83          printf("<div style='float:right;'><div style='margin-top:5px;margin-right:10px;'>Fundet solution med PID: %d</div></div>", solutionPID);
84      }
85
86      printf("</div>");
87
88      // Content
89      printf("<div id='content'><br />");
90
91
92      // Status
93      if(sendRequest == 1 && solutionPID != -1)
94      {
95          kill(solutionPID, SIGUSR1); // Starts request!
96          printf("<script>window.location='?success';</script>");
97      }
98      else if(sendRequest == 2)
99      {
100         printf("<br /><center><img src='/loading.gif'/></center><br />");
101     }
102
103     // Resultats
104     if(sendRequest != 3)
105     {
106         if(sendRequest != 0 && sendRequest != 2) // pwetty formatting
107         {
108             printNewlines(2);
109         }
110
111         printf("<b>Tidligere resultater:</b><br />");
112         DIR *d;
113         struct dirent *dir;
114         d = opendir("/home/pi/wasp/results");
115         if(d)
116         {
117             printf("<ul>");
118             while ((dir = readdir(d)) != NULL)
119             {
```

```
120                 if(strcmp(dir->d_name, ".") != 0 && strcmp(dir->d_name, ".."))
121                 {
122                     printf("<li><p><a href='?%s'>%s</a></p></li>", dir->d_name, dir↩
                            ->d_name);
123                 }
124             }
125             printf("</ul><br />");
126 
127             closedir(d);
128         }
129         printf("<br /><br /><b>Kendte noder:</b><br />");
130         printf("<ul>");
131         std::map<int, char *>::iterator it;
132         for (it = nodeMapping.begin(); it != nodeMapping.end(); it++)
133         {
134           printf("<li><p>Node %d, med navn '%s'</p></li>", it->first, it->second);
135       }
136       printf("</ul><br /><br />");
137     }
138     else // Åbner fil!
139     {
140       // Læs og print resultater
141         char path[200] = "/home/pi/wasp/results/";
142         strcat(path, openFileName);
143 
144         FILE *resFile;
145         resFile = fopen(path, "r");
146 
147         printf("<a href='/cgi-bin/WASP'>&larr; Tilbage</a><br />");
148         printf("<br /><b>Resultater fra '%s'</b><br /><br />", openFileName);
149 
150         if(resFile != NULL)
151         {
152             printf("<table cellspacing='0'><tr><td width='200px' style='border-↩
                        bottom: 1px solid black;'><b>Node</b></td><td style='border-bottom↩
                        : 1px solid black;'><b>Værdi</b></td></tr>");
153             int node = -1, value = -1;
154             int even = 0;
155             while(fscanf(resFile, "%d:%d", &node, &value) != EOF)
156             {
157               // Farve på row i tabel
158                 char *style;
159                 if(even == 0)
160                 {
161                     style = "";
162                     even = 1;
163                 }
164                 else if(even == 1)
165                 {
166                     style = " style='background-color: rgb(228, 228, 228);'";
167                     even = 0;
168                 }
169 
170                 // Value!
171                 char *val;
172                 val = (char *)malloc(15*sizeof(char));
173 
174                 if(value == -1)
175                 {
176                   val = "Ingen modtaget";
177                 }
178                 else
```

```
179                 {
180                   sprintf(val, "%d", value);
181                 }
182 
183                 printf("<tr><td%s>%s</td><td%s>%s</td></tr>", style, nodeMapping[↩
                        node], style, val);
184             }
185             printf("</table>");
186 
187             fclose(resFile);
188         }
189         else
190         {
191             printf("<p>Ingen resultater her!</p>");
192         }
193     }
194 
195     printf("</div>");
196 
197     // Footer
198     printf("<div id='footer'><p>WASP - SW513E15</p></div>");
199 
200     if(sendRequest == 2)
201     {
202         printf("<script>var interval = setInterval(testFile, 1000); function ↩
                    testFile() { var xhttp = new XMLHttpRequest(); xhttp.↩
                    onreadystatechange = function() { if (xhttp.readyState == 4 && xhttp.↩
                    status == 200) { window.location='?'+xhttp.responseText; } }; xhttp.↩
                    open('GET', '/ping.txt', true); xhttp.send(); } </script>");
203     }
204 
205     // Indhold slut (Lav footer fil)
206     printf("</body></html>");
207 
208     return 0;
209 }
210 
211 // Important stuff.
212 void printNewlines(int num)
213 {
214     for(int n = 0; n < num; n++)
215     {
216         printf("<br />");
217     }
218 }
```

```
1  // Create packet
2  Packet checkPacket(Data, 1, 2, 3, 4, 5, 6);
3
4  if(checkPacket.checksumMatches())
5  {
6      std::cout << "Packet is valid after initializing" << std::endl;
7  }
8
9  // Change values and check if packet is still correct
10 // PacketType
11 checkPacket.packetType = DataRequest;
12 if(!checkPacket.checksumMatches())
13 {
14     std::cout << "Packet is incorrect after changing packetType" << std::endl;
15 }
16 checkPacket.packetType = Data;
17
18 // Addresser
19 checkPacket.addresser = 2;
20 if(!checkPacket.checksumMatches())
21 {
22     std::cout << "Packet is incorrect after changing addresser" << std::endl;
23 }
24 checkPacket.addresser = 1;
25
26 // Addressee
27 checkPacket.addressee = 1;
28 if(!checkPacket.checksumMatches())
29 {
30     std::cout << "Packet is incorrect after changing addressee" << std::endl;
31 }
32 checkPacket.addressee = 2;
33
34 // Origin
35 checkPacket.origin = 1;
36 if(!checkPacket.checksumMatches())
37 {
38     std::cout << "Packet is incorrect after changing origin" << std::endl;
39 }
40 checkPacket.origin = 3;
41
42 // Value 1
43 checkPacket.value1 = 1;
44 if(!checkPacket.checksumMatches())
45 {
46     std::cout << "Packet is incorrect after changing value1" << std::endl;
47 }
48 checkPacket.value1 = 4;
49
50 // Value 2
51 checkPacket.value2 = 1;
52 if(!checkPacket.checksumMatches())
53 {
54     std::cout << "Packet is incorrect after changing value2" << std::endl;
55 }
56 checkPacket.value2 = 5;
57
58 // Value 3
59 checkPacket.value3 = 1;
60 if(!checkPacket.checksumMatches())
61 {
62     std::cout << "Packet is incorrect after changing value3" << std::endl;
```

```
63 }
64 checkPacket.value3 = 6;
65
66 // Test again with default values
67 if(checkPacket.checksumMatches())
68 {
69     std::cout << "Packet is correct with default values" << std::endl;
70 }
```

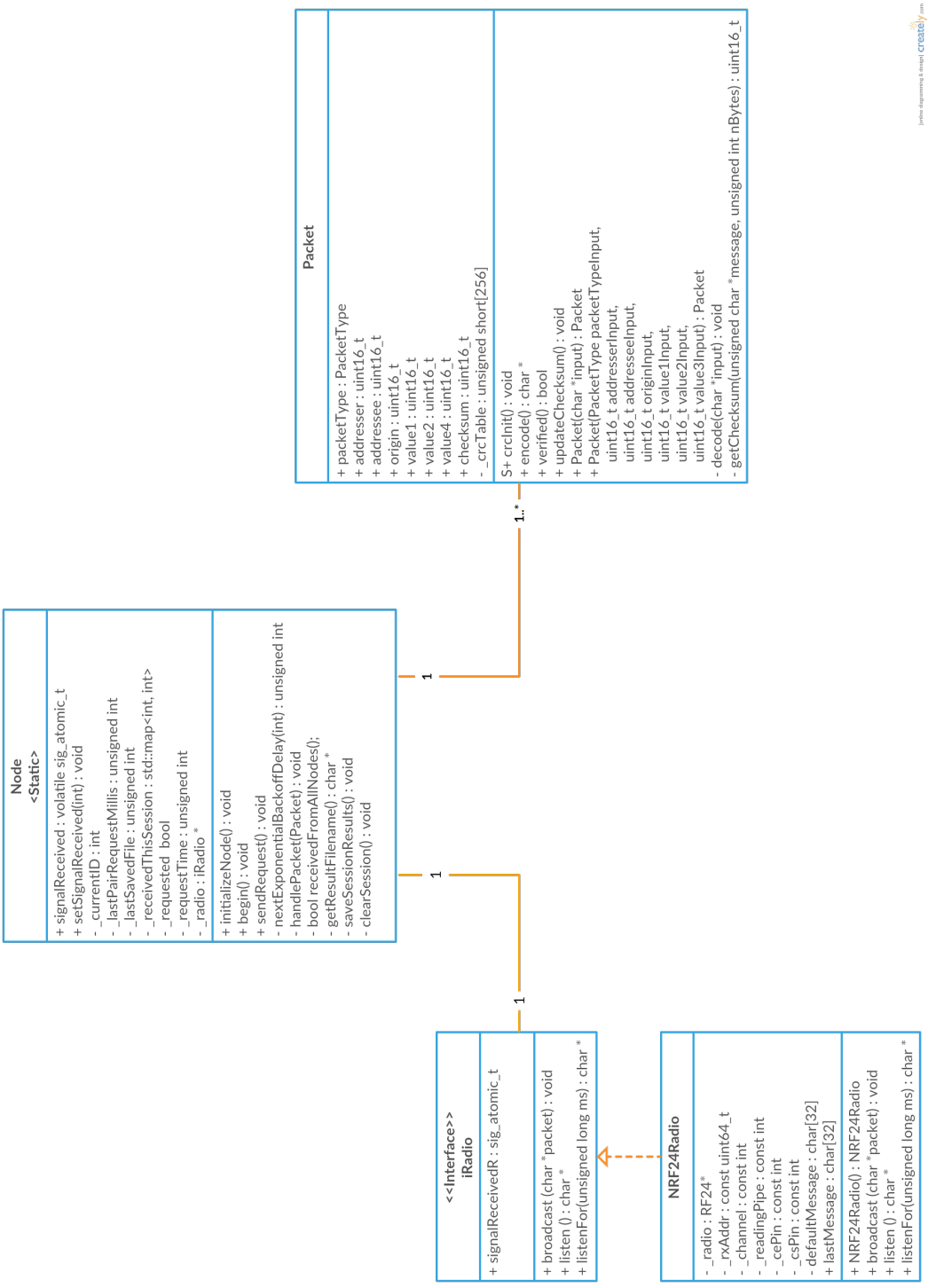

Figure E.1: Main node UML diagram.

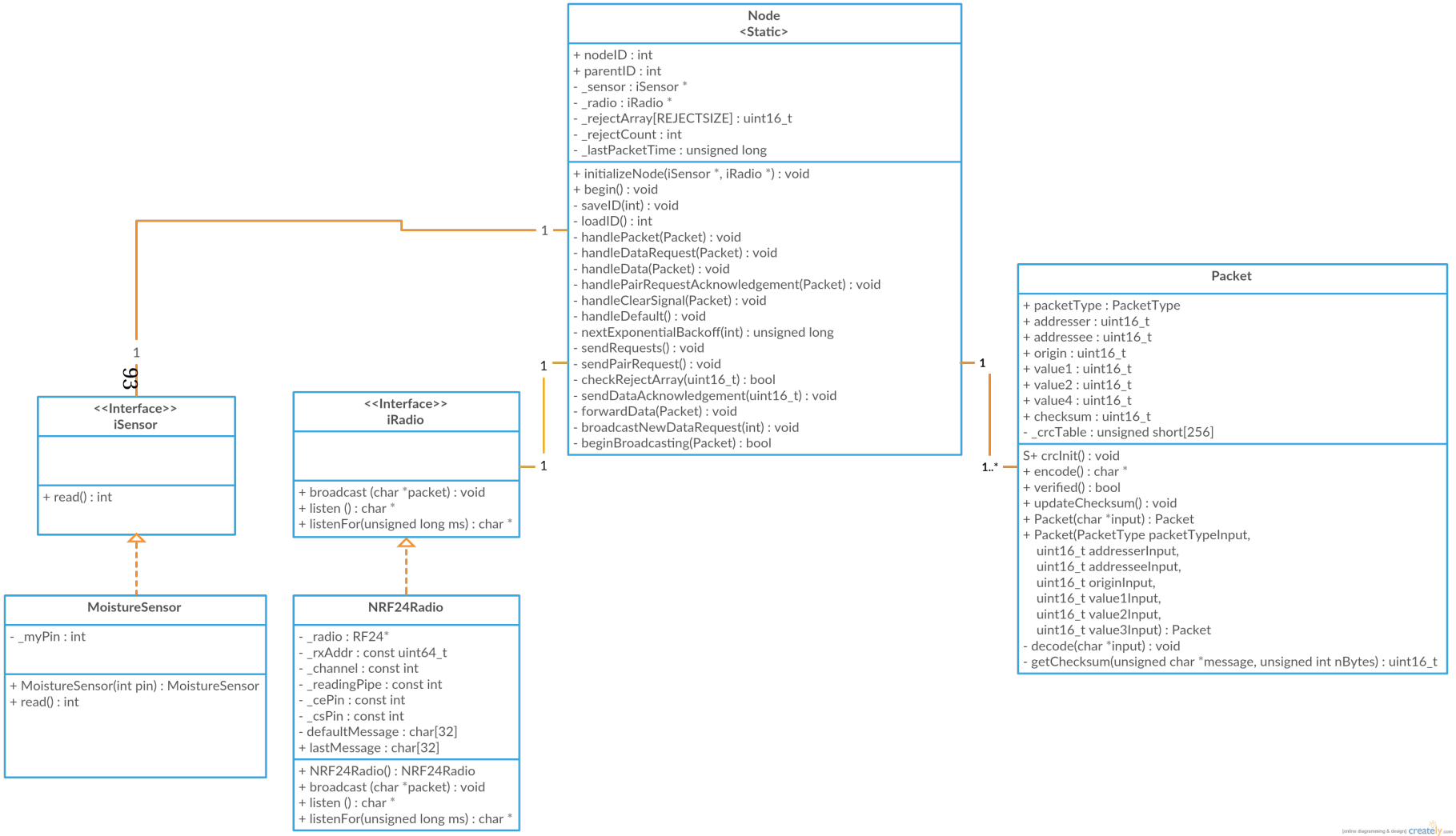

Figure E.2: Sensor node UML diagram.

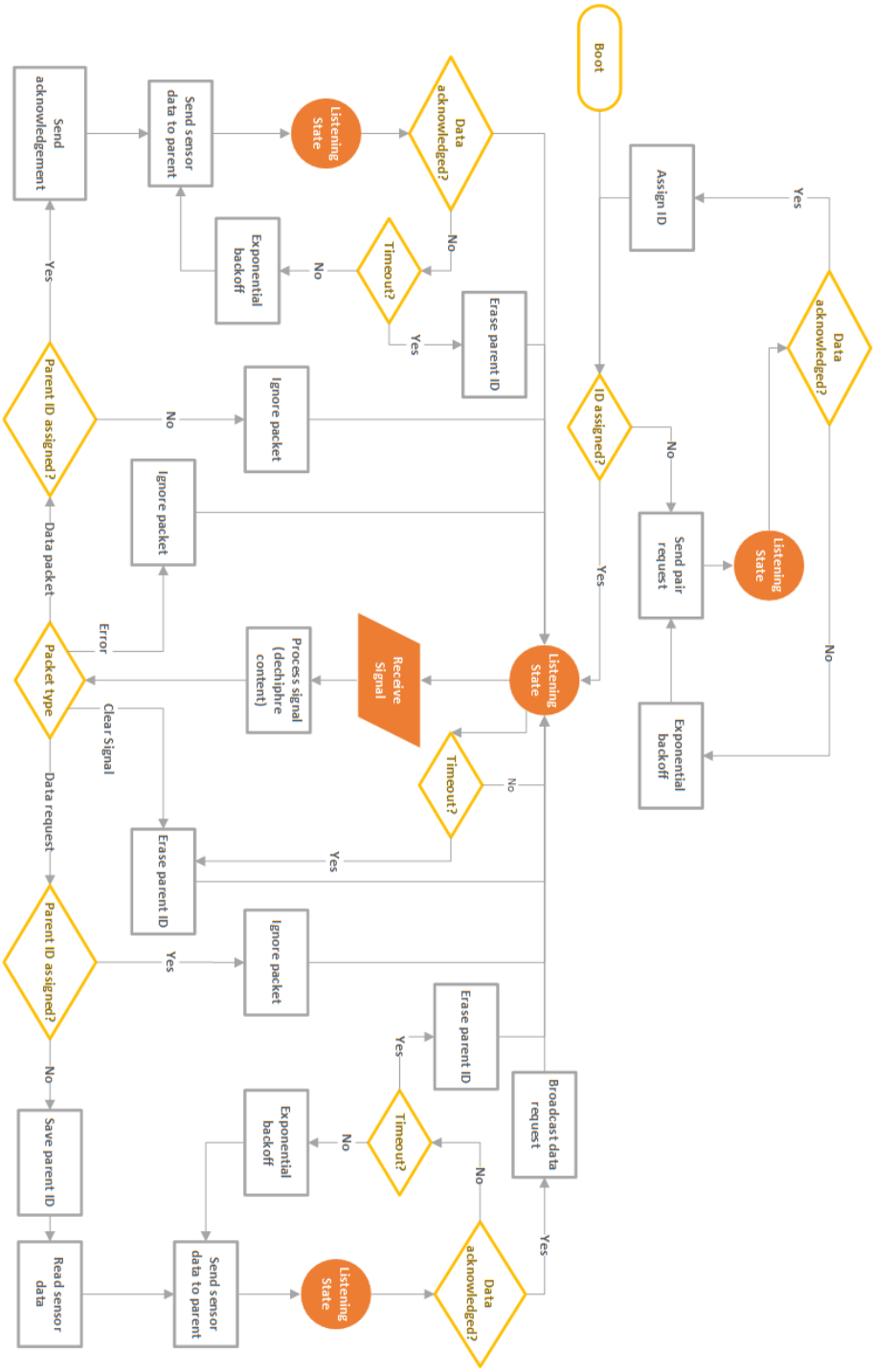

Figure F.1: Main node UML diagram.