Vehicle to vehicle communication

Alexandre GALLAND

Supervisor: Asiya KHAN
Tutor: Luc JAULIN
Address: Plymouth University

October 11, 2016

Contents

Remerciements

Fisrt of all, I would like to thank the all the people with who I worked in the Plymouth University.

I specifically thank my supervisor for offering me this opportunity to work in this University.

I thank Bob Bogard aswell, who greatly helped me by providing me with tools to construct the buggies.

Chapter 1

Context of the internship

This internship was done in the School of Marine Science and Engineering in the Plymouth University, in England, and with the help of the autonomous marine system research group. This research group's mission is to provide industries with autonomous operated systems, to conduct operations in littoral regions. A workshop was dedicated to this research group, and to the Control Club, which is a student club. However, this club was controlled by the lecturers, and some of the projects were done as courses, whereas some other were only recreational. The aim of this club was to make the students curious to the robotics. However, because the supply of this workshop was limited, especially the supply used for the electronics (such as wire or connectors), it was possible to buy some in an other department.

The subject of my internship was initially composed in two parts : Firstly, to further develop the buggies built by the Control club and demonstrate them as part of the UK Robotics week. Secondly to investigate the suitability of the Raspberry Pi as a micro controller for wireless communication in vehicle to vehicle communication. The second part of the subject was the most important, because more and more projects were using multiple autonomous vehicle at once (for example a collision avoidance system for marine vehicles).

Furthermore, I had to create one or two basic buggies, in order to create a manual for first year student who will have to build a similar buggy, and write scripts to accomplish a specific mission. Because those buggies might have to communicate to one another, the communication protocol I had to establish had to be easy to use, and independent of any scripts they will need to produce.

Chapter 2

The construction of a buggy

The buggies initially designed by the Control Club were using PIC micro-controller, as well as ultrasonic range detector, and a 433 MHz transceiver module.

The UK Robotics week finished, I received two Raspberry Pi 3 Model B, with the Raspbian Jessie Operating System (OS).

The first point to settle is the specifics the network will need. Ideally, this network should work as if every robots connected to it are cars in a road traffic. This means the robots only knows the other robots close to him. Then, none of them know the size of the network, or the number of nodes forming it. Neither does it knows the shape of the network. When an information has to be sent to another robot or group of robot, it does so by sending the information to its neighborhood, which will then relay it to their neighborhoods, until the recipient receives it. Furthermore, it is possible that multiple networks exists, without being connected to each other. Finally, the shape of the network is dynamic, which means each robots might have different neighborhood at different instant.

Then I had to imagine a buggy which will use certain electronic component in a pack, provided by the University. Those packs were initially packs used to create a buggy controlled by a PIC. I needed to update the components of a pack to create a buggy controlled by a Raspberry Pi 3.I had to change as few as possible components of the pack, and keep the price of a pack as low as possible. For this reason, I kept the same motors, which were RVFM DC motors. I added two components, one L298N dual motor controller module, and one UBEC Step-Down Converter (3V output). The basic electronic components (such as resistors) previously used were changed aswell. In the end, the components in a package were :

- Two DC motors with their wheels
- One motor controller module
- One Raspberry Pi 3 with a SD card and its protective case
- One HC-SR04 Ultrasonic Range Detector (URD)
- One UBEC

- One 9V battery
- Four AA battery with a batteries holder
- One breadboard
- Three 1kOhm resistors
- One red LED and one green LED
- Two battery-connectors

All additional components, such as wire or screw, could be obtained separately. Notice in the list above that there is no communication modules. The reason is that the Raspberry Pi 3 already has a Wi-Fi module in it. The electronical components were linked as shown in Figure 2.

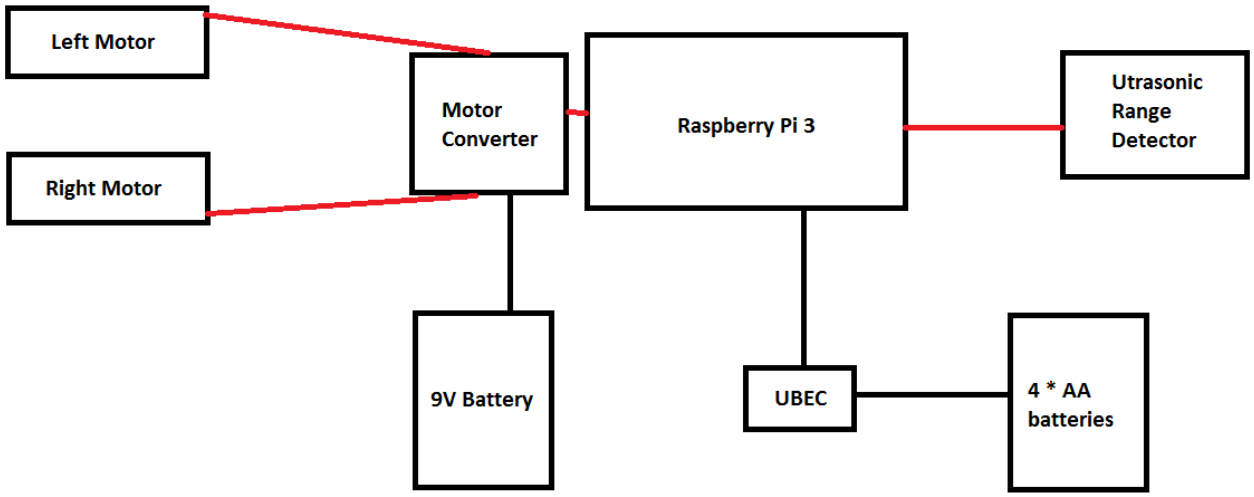

Figure 2.1: Electronical architecture of a buggy

The four AA batteries supply the power to the Raspberry, with the help of the UBEC. Indeed, the Raspberry Pi 3(RPi) needs exactly 5V to work, and around 2.5A. It reaches these figures with four AA batteries. The motor converter could have been used to supply the Raspberry with power, but the amperage was not high enough to make the Raspberry work properly. Then, a 9V battery is enough to power both motors. The main reason we use simple batteries is for mobility. They occupy little space, and are not expensive.

To create the structure of the buggies, I used 3D printers provided by the schools. This solution is not expensive, and you can use SolidWorks to create a scheme of the part you want to create, and "print" it easily. But a drawback is that it takes few hours to print, you might then wait if some people already plan to print something before you. The structure is composed of two stages. The lower one contains the motors and the motor controller, whereas the higher one is composed of the RPi, the breadboards, and the URD.

The reason we use resistors in this buggy is to lower the voltage going into the RPi via a GPIO input. Indeed, the RPi does not support more than 3.3V in input. When you plug the ECHO pin of the URD in the RPi, its voltage is equal to 5V. Then, we need to use a voltage divider as represented in Figure 2.

For the rest of this report, we will assume that the URD is connected to the RPi via the pins shown in the Figure 2, and the converter is connected as shown in the Figure 2.

2.1 Writing the basic scripts

This way, it is possible for anyone to build a similar buggy easily, which is cheap (the most expensive component is the Raspberry). After building it, some scripts needs to be written to make its use easy.

At this point, I had to decide what needed to be done to make its usage easier. The Raspberry Pi had the Raspbian Jessie's Operating system. Because some people who needs to use a buggy might not be familiar with Linux, I had to think of a way to use as few as possible command lines. Furthermore, the programming language used for the scripts should be one easy to master, which require as few effort as possible to use. I chose python. You can easily compile a python scripts, the syntax is simple and clear, and it is simple to master, compared to C and its pointer. Furthermore, you can find a lot of help on the internet, as well as a lot of libraries, enabling you to extend the range of the possibilities. As text editor, I thought about Gedit. It is a simple text editor, then easy to use, and easy to download. Furthermore, if you want more functionalities, you can download plugins, such as the terminal.

To give some examples of how to use the basic functionalities of the buggy, I wrote a script, containing the basic functions necessary in order to use the motors, as well as giving the distance found by the range detector.

The package RPi.GPIO is necessary here to control the input and the output.

Whenever you use the RPi's pins, you have to declare two things:

- The pin's number scheme you will use in your script

  You can choose two modes : the board mode, which assign the position of the pin as pin number. It goes from the center to the exterior, and from back (where you put the SD card) to the front. Or you can choose the BCM system. The function to use to assign a mode is setmode(GPIO.BOARD) or setmode(GPIO.BCM) from the library RPi.GPIO.

- The status of each used pins. Use the function setup(int pin, GPIO.IN) if the pin's

number equal to pin is an imput, or setup(int pin, GPIO.OUT) if it is an output. This function is provided by the library RPi.GPIO.

To control the motor, Pulse-Width Modulation (PWM) signals are used. Those signals are periodic. During a period, the amplitude is either maximum (here 3.3V) or minimum (0V). The duty cycle determine the proportion of time where the amplitude is maximum, to the period, in percentage. A duty cycle equal to 100 is then a constant signal with the maximum amplitude. To control one motor, two PWM signal has to be sent to the controller. The first one is used to rotate the wheel in one direction, the second one is used for the opposite direction. The higher the duty cycle is, the higher the rotation speed will be. To control both motors, the L298N converter then needs four signals going through IN1,IN2,IN3 and IN4. IN1 and IN2 control the left motor, and IN3 and IN4 control the right one. A low level function used in python is p = PWM(int pin, int frequency), where pin is the output pin where the signal is sent, and frequency the frequency of the signal. This function creates a PWM object, called here p. To choose the duty cycle of the signal, use the ChangeDutyCycle(int your_duty_cycle) function on the object p.

To use the ultrasonic range detector, the code written is as in Figure 2.1.

After setting up the pins, the HC-SR04 range detector needs a pulse through the "TRIG" pin to trigger the module, lasting at least 10 microseconds. Then, we calculate the time needed to sense a signal coming from the "ECHO" pin. We can then determinate the distance between the sensor and an potential obstacle, using the Equation $distance = timeWaited * 17150$. This formula consider that the sound speed in the air equals 343m/s, and will return a distance in cm.

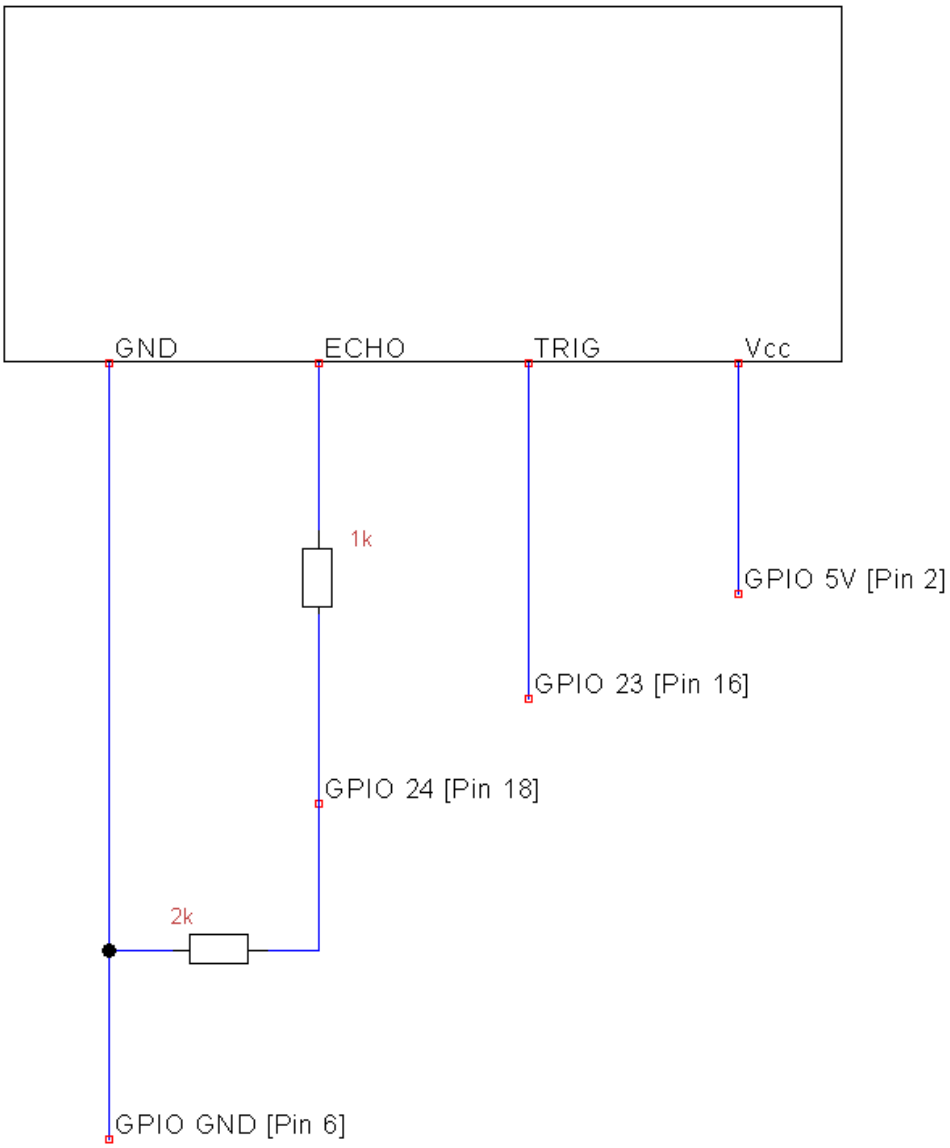

Figure 2.2: The voltage divider

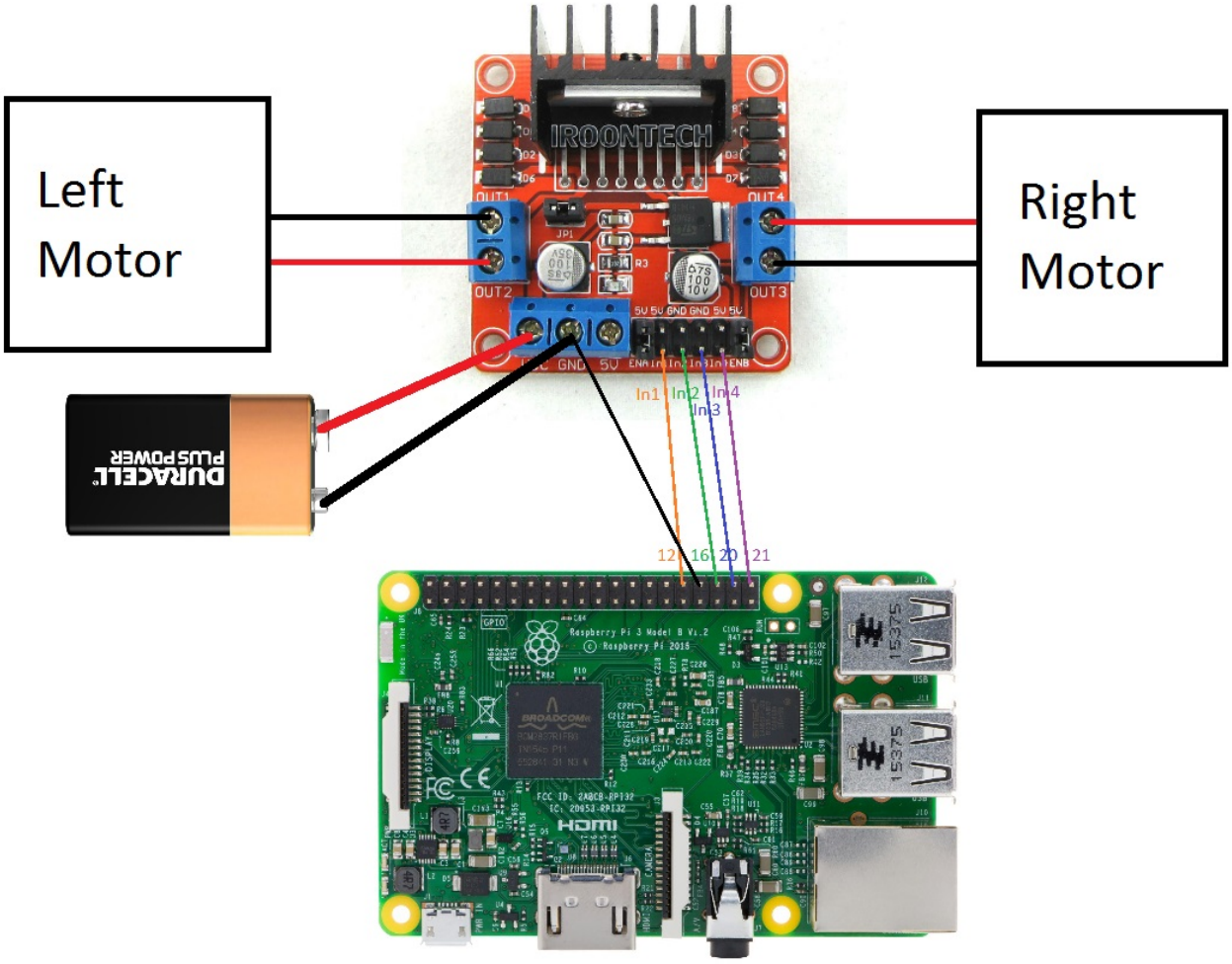

Figure 2.3: The connections of the motor controller

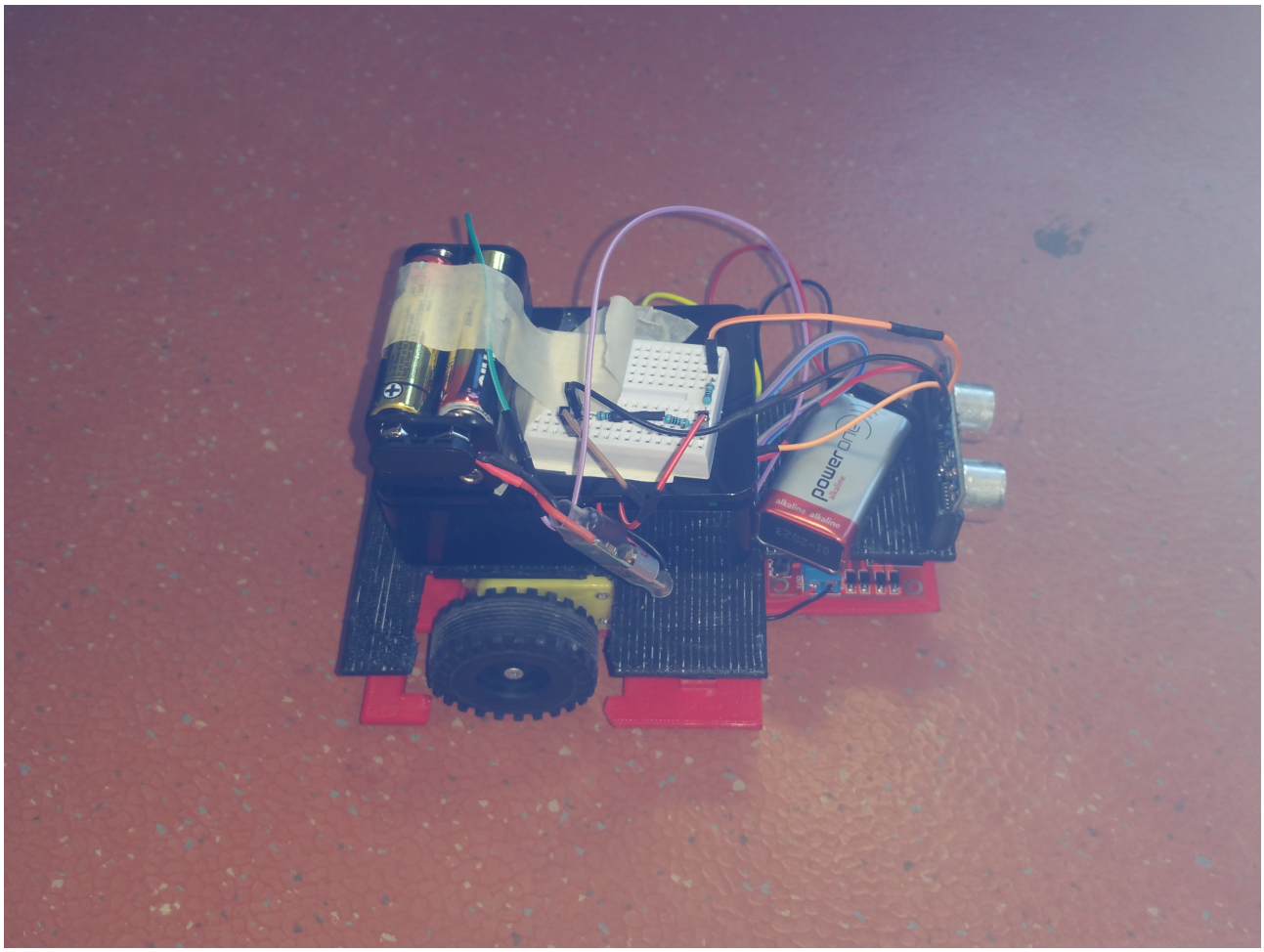

Figure 2.4: The buggy once finished

```
def findObstacle():
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(TRIG, GPIO.OUT)
        GPIO.setup(ECHO, GPIO.IN)

        GPIO.output(TRIG,GPIO.HIGH)
        time.sleep(0.2)
        GPIO.output(TRIG,GPIO.LOW)
        pulse_start=time.time()
        print "Searching for echo... "
        while GPIO.input(ECHO)==0:
            pulse_start=time.time()
        print "Echo found"
        pulse_end=time.time()
        while GPIO.input(ECHO)==1:
            pulse_end=time.time()
        pulse_duration =pulse_end-pulse_start
        distance = pulse_duration*17150
        distance=round(distance,2)
        print "Distance : ",distance,"cm"
        return distance
```

Figure 2.5: The code to get the range

Chapter 3

The communication protocol

After now that two basic buggies were created, I had to create a vehicle to vehicle communication protocol. As explained in the chapter 1, the network were the buggies would ideally work is dynamic, and unknown to each buggy. The two main communication protocol used by computer to communicate are TCP standing for Transmission Control Protocol, and UDP standing for User Datagram Protocol.

The first protocol is the most reliable one. A connection is firstly properly established between the communicating entities. The one wanting to communicate is called the client, and the other one is called the server. Then, the data can be transferred from the client to the server. Finally, the connection is properly closed. This protocol is secured, but works only if the connection is not interrupted, which might happen in a dynamic network.

The second protocol is simpler. It needs an IP address and a port. The entity which wants to send data simply send the data to the address via the selected port. It does not need to establish a connection with the receiver, nor does it needs to close it. Because of that, the sender does not know if the recipient received the message, or if the message was intercepted by another entity. This protocol is perfectly fit in our case, where the network changes all the time.

However, for a buggy to communicate with another buggy, it has first to get an idea of the network. Let us assume first that the network does not change with the time, and is as shown in Figure 3.

3.1 How each buggy send data

If two buggies not directly connected to each other have to communicate, they will send their message to their neighborhood, which will relay it to its other neighborhood, until the recipient receives the message. To do so, each buggies has to know the "distance" separating it to every other buggies of the network. What we call the "distance" between two buggies, is the minimum number of buggies required to transmit the message, from the creator of the message, to the final receiver. This means that two buggies directly connected have a distance of

0. This information is useful for the intermediate buggies that will have to relay the information. Indeed, let us assume that the buggy "A" wants to communicate with the buggy "F". A will send a message, containing the address of the recipient, and the distance separating each other, to all its neighborhood. Those neighborhood will check if they are the recipient, or if the distance in the message is zero (or less). If both criteria are false, they will send the message to their neighborhood, with a distance lowered by one. If they are the recipient, they save the message. This way, a message can be transferred through the network, without echoing indefinitely in the network. However, if the message is not received by the recipient, "A" won't have any way of knowing it. Two process are used to transfer the messages from one buggy to the other :

- send_data/send.py

  This process will send the message you want to send once. Before sending it, it will check the beginning of the message to see if there is a valid header, and will write the "distance" between itself and the receiver.

- send_data/recv.py

  This will receives every messages, check if it is the final recipient, and send it back to the network if it is not the case, lowering by one the distance written in the file.

Those two files can be found in the annex.

3.1.1 send.py

For this scripts and for the recv.py, the port used to transmit message is 9999. Furthermore, the messages have the form of text file.This text file is called msg.txt and is created in the folder send_data.

First, the script create the UDP socket, using the function socket(AF_INET, SOCK_DGRAM). Then, it tells the socket that the recipient in the socket is the broadcast. This means that every text files sent using sendto() function via this socket will be sent to every buggies directly connected to it. The sendto(string file_to_send, string addr) send the file named file_to_send to the address addr. In our case, even if we configured the socket to broadcast, we have to choose the address ("255.255.255.255",9999), 255.255.255.255 being the IP address for the broadcast, and 9999 the port. But before sending any file, the script search in status.txt the distance associated to the recipient, with the function searchIpStatus(). This status file can be found in the mapping folder. It contains every IP address in the network, followed by their distance to this buggy.

If the returned distance is zero or higher, it adds in the beginning of the message the message "Message for :" in the first line, and in the second line, the IP address of the recipient,one space, and the distance, using the function checkFormatToFile().

After this, the message is sent using the function sendData(). In this function, the file to send is cut into packets, which are sent one after the other. The socket is then closed.

3.1.2 recv.py

This script receives all the messages, and send it back if necessary.

The code is then split in two parts : when the RPi receives a message, handled by receiveMsg(), and when it send it back to the network, using transmitMsg().

ReceiveMsg first create a UDP socket, binded to the port 9999. Whenever it receives a packet of a message, it updates a table containing all the IP address which send data to this buggy. This way, by clearing it every time we use the receiveMsg function, we can see if multiple buggies sent a message while receiveMsg was running. Furthermore, everytime it receives a packet, it checks that the address of the sending buggy correspond to the address of the buggy which send the message which is being recomposed. Or else, if multiple message are sent at the same moment, they will all mix into one big message, which will have only one recipient. The received message is then saved under the name recv.txt in the folder send_data.

Once the message is fully recomposed, the script test if it needs to send it back to the network. It does so if the distance written in the received file is higher than 0, and if the recipient is not him. It is sent back to the network exactly the same way than send.py does.

3.2 How each buggy acknowledge the network

Initially, no buggy has an idea of the network it is in. Two scripts will then run, in order to find the relative distance between a buggy and the other buggies in the network. The way these distances are known is explained in the Figure 3.2.

To do so, each buggy send to its neighborhood a file, called “status.txt”, in the beginning of each iteration. In this file, initially empty, will be recorded all the buggies’ IP address it received a status file from. This means that when a status file is received by a buggy, it will see the IP address of the sender, and log it in its own status file with an associated distance equal to 0 (if it was not allready written in the file this IP address with a lower distance). Then it will compare the received status file with its own one. For every unknown IP address found in the other status file, it will add this IP address into its own status file, and associate to it the same distance than in the other file, plus one. For every known IP adddress, it will compare the distance written in each status file, after having added one to the distances in the received status file, and will keep the lowest one. It will then repeat all those steps indefinitely. Every once in a while, all the status file are emptied, or else any change in the network might not appear.

Two scripts handle this algorithm:

- mapping/recvfilebradc.py

  This scripts receives all the sent status files on the port 54545, and analyses them to modify its own status file.

- mapping/sendfilebradc.py

  This script send every half second its status file in broadcast, on the port 54545.

Those two files and recv.py are launched using one script, called “network.py”.This file uses the function os.system() for each scripts. This function can launch any Linux command typed in the brackets. Here, we launch each time a terminal which will display pieces of information concerning its script. This is done with the command “ gnome-terminal -e ’bash -c \"python ./your_script.py; exec bash\"’ “. gnome-terminal creates a terminal, which will launch in it the script “your_script.py”.

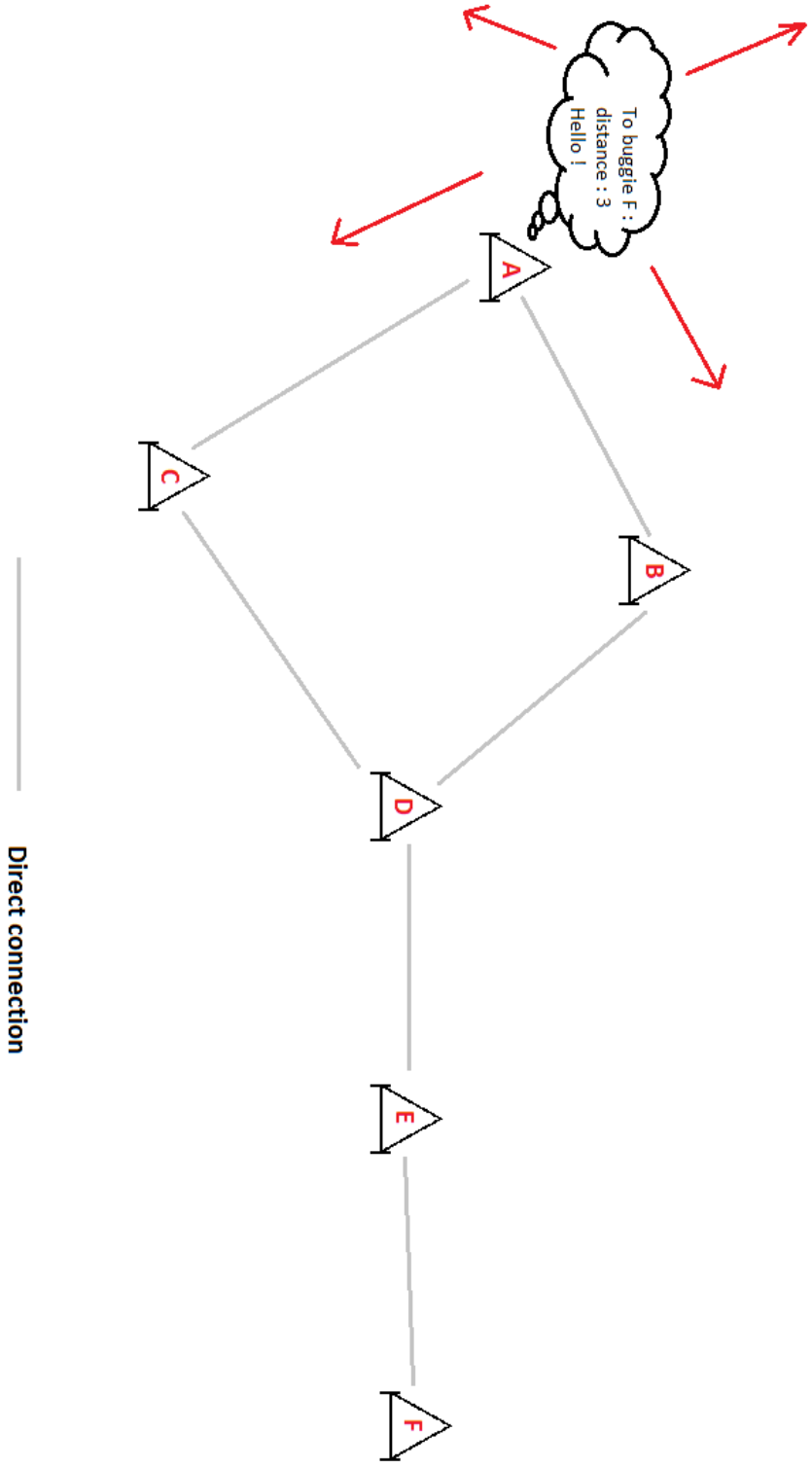

Figure 3.1: The chosen example of the network

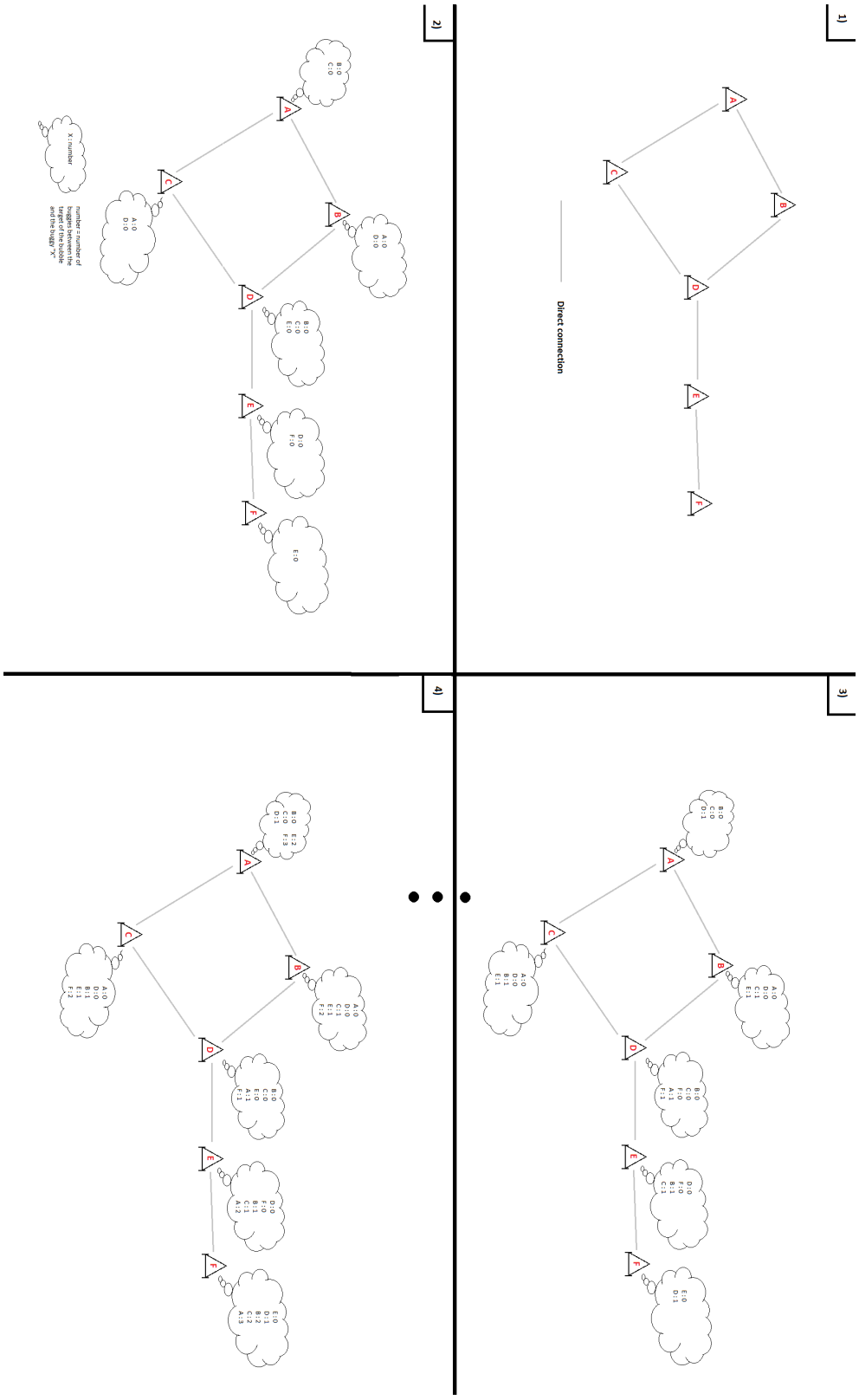

Figure 3.2: How to get the distances

Chapter 4

Results

At the end of the internship, two buggies have been built, and were working quite well. The communication system could only be tested with a network composed of two buggies. Even though it was working well, some issue might appear with a bigger network. During this internship, I learned a lot about different communication protocols. In helped me train my English aswell.