M.Sc. Thesis
Computer Science and Engineering

DTU Space
National Space Institude

Relative positioning system for UAVs in swarming applications

Andrés Cecilia Luque (s146537)

Kongens Lyngby 2018

DTU

DTU Space
National Space Institute
Technical University of Denmark

Elektrovej, buildings 327, 328, 371 and Ørsteds Plads, building 348
2800 Kongens Lyngby, Denmark
www.space.dtu.dk

Abstract

During the past decade we have seen an increasing number of civilian applications for Unmanned Aerial Vehicles (UAVs), which have boosted the amount of efforts invested on research and development. One of the main challenges is the operation of multiple robots at the same time: commanding a swarm of UAVs. But, how to accurately position each one of them?

This master thesis designs and implements a decentralized solution that allows each UAV in a swarm to calculate its position relative to the position of the neighbours around, without the assistance of an external system, and using open source hardware and software.

Preface

This master thesis was prepared at the Denmark's National Space Institute, at the Technical University of Denmark, in fulfillment of the requirements for acquiring a master degree in Computer Science and Engineering.

Kongens Lyngby, September 2, 2018

Andrés Cecilia Luque (s146537)

Acknowledgements

I would like to thank here the people that contributed to this master thesis. Thanks Xiao for your feedback, your positiveness and support during the last months. It has been very valuable, specially when things were not happening as I planed them. Thanks the whole team at Bitcraze, specially Kristoffer, for your support. Thanks for borrowing me the quadcopters and hardware I needed, for welcoming me in your office and for your ideas and feedback. Your input had a great influence in the development, helping me find solutions to problems when I run out of ideas. I also want to thank my roommates, that did not contribute to this work but have stand, supported and cooked for me during the hardest weeks (probably, the only reason I am still alive is because of that).

This document also closes a chapter in my life. I would like to dedicate it to the people that has been with me during the last two years:
To my family, always supportive, even when we are thousands of kilometer away.
To my friends in Copenhagen, for the memories we already created, and the adventures about to come.
To Saul, because the time in Singapore was a blast, and that was just the start.
To my distant friends, who despite of how much time passes, are always there.

Finally, I would like to start this master thesis with some sentences from the book "The Opposite of Loneliness":

> *We don't have a word for the opposite of loneliness, but if we did, I could say that's what I want in life. What I'm grateful and thankful to have found at Yale, and what I'm scared of losing when we wake up tomorrow after Commencement and leave this place.*
> *It's not quite love and it's not quite community; it's just this feeling that there are people, an abundance of people, who are in this together. Who are on your team. When the check is paid and you stay at the table. When it's four A.M. and no one goes to bed. That night with the guitar. That night we can't remember. That time we did, we went, we saw, we laughed, we felt. The hats.*
> *...*
> *But let us get one thing straight: the best years of our lives are not behind us. They're part of us and they are set for repetition as we grow up and move to New York and away from New York and wish we did or didn't live in New York.*
> *...*
> *We have these impossibly high standards and we'll probably never live up to our perfect fantasies of our future selves. But I feel like that's okay.*
>
> *We're so young. We're so young. We're twenty-four years old. We have so much time.*

Contents

List of Figures **xi**

List of Tables **xiii**

List of Code Blocks **xv**

List of Acronyms **xvii**

1 Introduction **1**

2 State of the art **3**
2.1 Indoor positioning: general overview 3
2.2 Indoor positioning: UWB signals 4
2.3 Ranging scheme for measuring the TOF 5
2.4 Position estimation technique 5

3 Development environment **7**
3.1 Hardware 7
3.2 Software 8
3.2.1 On-board software 8
3.2.1.1 The building process 8
3.2.1.2 The software components 10
3.2.1.3 The booting sequence 12
3.2.1.4 The flashing sequence 12
3.2.2 The STM Firmware 13

4 Problem breakdown **15**
4.1 Challenges when designing the ranging scheme 15
4.2 Challenges introduced by the development platform 17
4.3 Challenges introduced by the communication channel 18

5 Design **19**
5.1 Trilateration: N-Dimension analysis 19
5.2 Ranging scheme 21
5.2.1 Existing ranging schemes 21
5.2.1.1 One to one ranging: TWR 21
5.2.1.2 One to one ranging: SDS-TWR 22
5.2.2 Many to many: swarm ranging scheme 23
5.3 Position estimation 25
5.3.1 Broadcast of TOF values 25
5.3.2 Generate a coordinate system 26
5.3.3 Position drifting 27

6 Implementation **29**
6.1 Packet identification . . . 29
6.2 Quadcopter identification . . . 30
6.3 Preventing duplicated packets: the sequence number . . . 32
6.4 Timestamping tx and rx events: DW1000 . . . 33
6.4.1 DW1000 limitations . . . 33
6.4.2 Implemented solutions . . . 34
6.5 Randomized transmission . . . 38
6.6 Ranging scheme: implementation . . . 40
6.7 *STATUS* packet content . . . 42
6.8 Position estimation: kalman filter . . . 45
6.8.1 Kalman filter refactoring . . . 46
6.8.2 Kalman filter: prediction step . . . 48
6.8.3 Kalman filter stabilization . . . 50
6.8.4 Final implementation . . . 50
6.9 Memory usage . . . 52
6.9.1 Used data structures . . . 52
6.9.2 Implementation using dynamic allocation: libdict . . . 53
6.9.2.1 Libdict: test implementation . . . 54
6.9.2.2 Libdict: test results . . . 55
6.9.3 Implementation using static allocation . . . 56
6.10 Memory alignment . . . 58
6.10.1 Unaligned access . . . 58
6.10.2 The packed attribute . . . 59

7 Results **61**
7.1 Position estimation accuracy . . . 61
7.2 CPU load and memory usage . . . 64
7.2.1 CPU load . . . 64
7.2.2 Memory usage . . . 65
7.3 Standard deviation adjustment . . . 67
7.4 Ranging frequency: performance . . . 69

8 Conclusions **71**

9 Future work **73**

Bibliography **75**

A Maximum throughput **79**

B Static VS non-static implementation of the kalman filter **81**
B.1 Storage . . . 81
B.2 API . . . 83

C Libdict test results **85**

D Storage: neighbour, TOF and context **89**

List of Figures

3.1 The plotter tab, in the crazyflie client 8

4.1 A reflected radio packet reaching the antenna 16

5.1 An example of trilateration in 2D 19
5.2 The packet exchange during the Two Way Ranging (TWR) ranging scheme . 21
5.3 The packet exchange during the Symmetric Double Sided Two Way Ranging (SDS-TWR) ranging scheme 22
5.4 The packet exchange during the swarm ranging scheme 23
5.5 The Time of Flight (TOF) values known/unknown by quadcopter A after a full ranging cycle, in full/dotted lines 25
5.6 The TOF values known by quadcopter A as the rest of the neighbours transmit their *STATUS* packets. In black, the TOF values resulting from the quadcopter transmitting. In gray the TOF values resulting from previous quadcopters transmitting 26
5.7 Coordinate system creation. In blue, the quadcopter. In green, the neighbours building the coordinate system. In red, a neighbour positioned based on the already built coordinate system 27
5.8 1D relative positioning of two stationary quadcopters, where one of them calculates slightly longer TOF than the other 28

6.1 Representation of four times the crazyflie was booted, where the seed was set in the same millisecond 30
6.2 Antenna delay diagram 34
6.3 Calculated clock correction between two quadcopters 35
6.4 Filtered clock correction between two quadcopters. The absolute top and bottom limits are out of the range of the vertical axis 36
6.5 Transmission events randomized inside a transmission window 38
6.6 First implementation approach for timestamp exchange 40
6.7 Final implementation approach for timestamp exchange 41
6.8 Introducing error in the measurement of d_{t-1} makes it impossible to calculate P_t. In dotted lines, the real distances. In full lines the measured distances. . 45
6.9 Process for updating the position of the quadcopters 51
6.10 Aligned 32 bits integer 58
6.11 Unaligned 32 bits integer 58
6.12 Reading an unaligned 32 bits integer 58
6.13 Normal struct memory layout 59
6.14 Packed struct memory layout 59

7.1 The scenario used to obtain the results. The XY plane in this figure represents an horizontal surface (a table), and all the quadcopters on it are placed on the surface (Z position = 0) 61

7.2 The four locations of the recorded data 61
7.3 Central Processing Unit (CPU) load for a quadcopter during normal operation, while running the swarm ranging scheme 64
7.4 1D relative position calculation for two quadcopters. The standard deviation value is 0 67
7.5 1D relative position calculation for two quadcopters. The standard deviation value is 0.25 68
7.6 Number of sent and received packets during normal operation 69

A.1 How the measurement of a successful or failed ranging attempt is implemented 79

List of Tables

2.1 Comparison of the different technologies used for indoor location 4

3.1 Flash memory layout for the NRF 11
3.2 Flash memory layout for the STM 11

6.1 Example calculations when the timestamp counter wraps around 34
6.2 Wrap around when using unsigned integers for calculations 36

7.1 Real VS estimated position values for location 1 62
7.2 Real VS estimated position values for location 2 62
7.3 Real VS estimated position values for location 3 62
7.4 Real VS estimated position values for location 4 63

A.1 Real measurement of the maximum throughput per second 80

C.1 Dynamic insert/search test 85
C.2 Dynamic insert memory benchmark (higher is better) 85
C.3 Dynamic insert speed benchmark (executed during 2000ms, higher is better) 86
C.4 Static insert/search test 86
C.5 Static insert/search test, reusing same key pointer but changing its value . . 87
C.6 Static insert speed benchmark (executed during 2000ms, higher is better) . . 87

List of Code Blocks

3.1 Sections information extracted from the crazyflie firmware binary by running the command `arm-none-eabi-readelf -S cf2.elf` . . . 9

6.1 Application Programming Interface (API) for generating quadcopter identifiers 31
6.2 The *STATUS* packet declaration . . . 42
6.3 The header declaration for the *STATUS* packet . . . 42
6.4 The payload declaration for the *STATUS* packet . . . 43
6.5 The function returning the estimated position, as implemented in the static kalman filter (`estimator_kalman.c` file) . . . 47
6.6 The function returning the estimated position, as implemented in the non-static kalman filter (`estimatorKalmanEngine.c` file) . . . 47
6.7 The non-static kalman filter engine (`estimatorKalmanEngine.h` file) . . . 47
6.8 An example of a call to the non-static kalman filter API . . . 48
6.9 API for accessing elements from both the neighbour and TOF storage . . . 57
6.10 Normal struct . . . 59
6.11 Packed struct . . . 59

7.1 Memory usage without the swarm ranging scheme . . . 65
7.2 Memory usage when configuring a maximum of 16 tracked neighbours . . . 65

B.1 Original code from the static kalman filter (`estimator_kalman.c` file) showing the variables that store the internal state . . . 82
B.2 Original code from the non-static kalman filter (`estimatorKalmanStorage.h` file) showing the variables that store the internal state . . . 82
B.3 Original code from the static kalman filter (`estimator_kalman.h` file) showing its API . . . 83
B.4 Original code from the non-static kalman filter (`estimatorKalmanEngine.h` file) showing its API . . . 84

D.1 The declaration of the struct used for storing neighbours data . . . 89
D.2 The declaration of the struct used for storing TOF data . . . 89
D.3 The declaration of the struct used for storing the internal state of quadcopter 90

List of Acronyms

ALOHA Advocates of Linux Open-source Hawaii Association.

AOA Angle of Arrival.

API Application Programming Interface.

CPU Central Processing Unit.

Crazyradio PA Crazyradio Power Amplifier.

FreeRTOS Free Real Time Operating System.

GPS Global Positioning System.

IDE Integrated Development Environment.

IMU Inertial Measurement Unit.

LOS Line of Sight.

LPS Loco Positioning System.

MBR Master Boot Record.

MBS Master Boot Switch.

MCU Micro Controller Unit.

NLOS Non Line of Sight.

NRF NRF51822 chip.

OTA Over the Air.

RAM Random Access Memory.

RFID Radio Frequency Identification.

RSSI Received Signal Strength Indicator.

SDS-TWR Symmetric Double Sided Two Way Ranging.

SPI Serial Peripheral Interface.

STM STM32F405 chip.

TDOA Time Difference of Arrival.

TOA Time of Arrival.

TOF Time of Flight.

TWR Two Way Ranging.

UAV Unmanned Aerial Vehicle.

USB Universal Serial Bus.

UWB Ultra Wide Band.

WLAN Wireless Local Area Network.

CHAPTER 1
Introduction

During the past decade we have seen an increasing number of civilian applications for UAVs. At the same time, the efforts invested on research and development are more than ever, with thousands of scientific papers being published [1].

The continuous evolution of technology opens the possibility for more intelligent robots. The use of sensors and on-board navigation systems allows a precise control, and the interaction with the environment: there are already several commercial examples of UAVs used for aerial inspection [2], or video recording [3]. For a UAV to be able to fly, it needs to get information about it's position and movement. For outdoor applications Global Positioning System (GPS), ultrasonic sensors and optical cameras are used to obtain information about location, altitude and obstacles around. Indoors, smaller UAVs are positioned using a variety of setups: for high precision, expensive optical systems with multiple cameras track the position of the UAVs and assist on the navigation [4][5]. For lower precision (in the order of 10cm), it is possible to implement a more affordable solution by setting up a system of transceivers that send signals between them and the UAV, measure the travel time and calculate the position using triangulation, trilateration, or similar algorithms.

Now that the technology is becoming more mature, one of the main challenges is the operation of multiple robots at the same time. While commanding a swarm of UAVs, it is necessary to avoid collisions between them. The positioning systems mentioned in the previous paragraph do support it, but they present several drawbacks:

- All of them rely on an external centralized system in charge of providing positioning information. This makes the swarm vulnerable, as the failure of such component results in the failure of the whole system. For example, the Intel's swarm could not take off in 2017, during a preview of their light show for the Singapore national day, due to lack of GPS signal [6].
- The affordable solutions usually require to setup anchors around the flying area in order to configure a coordinate system, limiting the scalability.
- The information about the position of the rest of UAVs in the swarm is not available on board in each of them: in case of the optical systems, a central controller knows the position of each UAV and commands them accordingly; in case of a system based on receivers, each UAV knows it's position, but there is no knowledge about the positions of all the others.

This master thesis designs and implements a decentralized solution that allows each UAV in the swarm to calculate its position relative to the position of the neighbours around, without the assistance of an external system, and using affordable open source hardware and software. Its contents are organized as follows:

- Chapter 2 presents an overview of the latest research in indoor positioning.
- Chapter 3 presents the platform to be used during development.
- Chapter 4 highlights several challenges of designing and implementing the relative positioning system.
- Chapter 5 presents the design of the ranging scheme and the position estimation technique.
- Chapter 6 presents the implementation of the ranging scheme and the position estimation technique, and the problems found during development.
- Chapters 7 and 8 present the results obtained and the conclusions of the author.
- Chapter 9 proposes several ideas for future work.

CHAPTER 2
State of the art

This chapter provides the reader with an overview of the latest advances in indoor positioning, focusing in Ultra Wide Band (UWB) solutions applied to UAVs.

2.1 Indoor positioning: general overview

There are multiple technologies and techniques used for indoor positioning. Several review papers made from 2015 to 2017 [7][8][9][10] offer a classified overview of the state of the art, highlighting the characteristics of each approach.

A first classification of indoor positioning systems divides them in two groups: device-based and device-free [8]. In device-based localization, a device is attached to the target and computes its position through cooperation with other deployed devices: for example, calculating the position of a tag relative to the anchors placed around. In device-free, the target does not carry any device, and is the deployed infrastructure the one calculating the target's location: for example, a set of cameras tracking a target, and using image processing to compute its position.

A second classification is done depending on the technology used: optical (using cameras), infrared, ultrasonic or radio. When using radio, the classification falls into Bluetooth, Radio Frequency Identification (RFID), UWB or Wireless Local Area Network (WLAN). Their main highlights are:

- Positioning based on cameras has a very wide range of accuracy and costs depending on the cameras used. For example, using a smartphone camera has low cost and easy setup, providing low-medium accuracy. Using a tracking system such as [4][5] has high cost and requires a complex setup, but provides a very high accuracy.
- Optical positioning based on cameras is the only positioning system classified as device-free, while the others are device-based.
- All of the indoor positioning systems require to deploy infrastructure around, except for WLAN, that uses the existing wifi infrastructure:
  - Optical positioning requires setting up the cameras.
  - Infrared, ultrasonic, Bluetooth, RFID and UWB require setting up transceivers or sensors.
- Optical and infrared technologies require Line of Sight (LOS) between the target and the sensors around: an obstacle will make positioning impossible. Ultrasonic and radio do not require LOS, being UWB and WLAN the best performers. In addition, UWB is specially robust against multipath effects.

The following table shows a sum up of their characteristics:

Technology	Accuracy	Cost	Requires infrastructure deployment	Interference resistance
Optical (using cameras)	Medium - Very High	Medium - Very High	Yes	Requires LOS. Susceptible to environment light changes
Infrared	Medium	Medium	Yes	Requires LOS. Susceptible to environment light changes
Ultrasonic	Medium	Medium	Yes	Susceptible to multipath effects
Bluetooth	Low	Low	Yes	Susceptible to signal interference
RFID	Low	Low	Yes	Susceptible to multipath effects
UWB	High	Low	Yes	Robust against multipath and NLOS
WLAN	Low - Medium	Low	No	Robust against NLOS

Table 2.1: Comparison of the different technologies used for indoor location

Indoor positioning of UAVs requires high accuracy. Thus, looking at the table is possible to understand why camera-based tracking systems are the most popular method when the cost of the system is not a limitation, and why UWB is a good alternative when looking at a more affordable setup.

2.2 Indoor positioning: UWB signals

UWB has become one of the preferred technologies for indoor positioning. This is due to its characteristics [11]:

- It can reach high data rates.
- Using high bandwidth and short pulses reduces the effect of multipath interferences. It also improves robustness against the noise introduced by other communication devices.
- The low frequency used by UWB allows the signal to pass through obstacles such as walls and objects (does not require LOS).
- The cost of UWB equipment is low.
- The energy consumption is low, compared to other technologies.

- UWB can accomplish high accuracy, with position errors in the order of centimeters.

When using UWB signals, there are several signal properties that can be measured in order to calculate the distance between transmitter and receiver. The main ones are [12]:

- Received Signal Strength Indicator (RSSI): the transceivers measure the signal strength, which suffers from attenuation during transmission. That attenuation is an indicator of the distance between the transmitter and receiver.
- Time of Arrival (TOA): measures the time of arrival of the signal, which is an indicator of the distance between the transmitter and receiver (it is also known as TOF).
- Time Difference of Arrival (TDOA): measures the difference of time of arrival between two transmissions coming from two different transmitters, which is an indicator of the position of the receiver between them.

It is also possible to use the Angle of Arrival (AOA) of the signal, but that is not considered in this classification because it requires setting up multiple UWB antennas per receiver.

2.3 Ranging scheme for measuring the TOF

The main interest of this master thesis is the TOF measurement. In order to obtain it, multiple packet exchanges are required. The algorithm that controls how this exchange is performed and what data should travel in the packets is called ranging scheme. The goal is to obtain the most accurate TOF measurements executing the minimum amount of transmissions. The most popular schemes are TWR and SDS-TWR, which will be analyzed in detail during next chapters. Also, modifications to this ranging schemes have been proposed recently for specific scenarios [13][14], but none of them seem to be as popular as SDS-TWR.

2.4 Position estimation technique

Once the TOF values are obtained, they need to be combined in order to compute the position. This is not a trivial task, as solving the trilateration equations while also handling measurement errors is complex, and has led to multiple research papers. Several reviews [15][16] show that the most popular solutions are:

- The analytical method + smoothing: this involves calculating the position by computing the nonlinear trilateration equations directly. Such calculation is not easy, and grows in complexity as the amount of dimensions increase (2D, 3D). In addition, it does not provide any protection against measurement errors. In order to make a better estimation, it is possible to apply temporal or spatial smoothing (average value of measurements within a sliding window).

- Least-squares method: this method is able to calculate an estimated position while able to handle measurement errors. The main issue is that, similarly to the analytical method, it requires all the measurements to be ready in order to perform the calculations.
- Bayesian filtering approaches: they are able to estimate the most probable position from multiple TOF measurements, even when they include error. They also have the benefit of being able to process one measurement at a time, giving a new and better estimation every time. The most popular of this approaches is the kalman filter.

The main interest of this master thesis is the usage of a kalman filter for positioning, which will be analyzed during next chapters.

CHAPTER 3
Development environment

This chapter provides the reader with an overview of the development platform in terms of hardware and software.

3.1 Hardware

The hardware platform used is the crazyflie 2.0[1]. Its approximate size is 10cmx10cm, and 3cm tall. It is manufactured specifically for development: all the hardware is open source, and the software is open source and free. Its main characteristic is that it provides expansion boards, that allows to add and connect new hardware components (their official name is decks) [17]. Its main specifications are:

- It mounts two Micro Controller Units (MCUs):
  - The STM32F405 chip (STM), used as main application MCU. It has a Cortex-M4 core running at 168MHz, with 192kb SRAM and 1Mb flash memory.
  - The NRF51822 chip (NRF), used for radio communications and power management. It has a Cortex-M0 core running at 32Mhz, with 16kb SRAM and 256kb flash memory.
- It mounts a MPU-9250 Inertial Measurement Unit (IMU), which includes a 3 axis accelerometer, 3 axis magnetometer and 3 axis gyroscope.
- It mounts a 20dBm radio amplifier (tested to work up to 1km range LOS). It is intended to be used with the Crazyradio Power Amplifier (Crazyradio PA), a Universal Serial Bus (USB) radio dongle designed for communicating with the crazyflie from a computer.

A complete list can be found in [18]. An overview of the hardware architecture can be found in [19].

In addition, to be able to measure distances, additional hardware is required. For this purpose, the Loco Positioning System (LPS) deck is used. It communicates with the main MCU by Serial Peripheral Interface (SPI). Its characteristics are:

- It is based on the Decawave DWM1000 module, a UWB transceiver operating in the range of 3.2GHz to 7GHz. The channel bandwidth is 500MHz.
- Under default configuration, it has a ranging accuracy of 10cm (can vary depending on the ranging scheme and the configuration of the DW1000).
- Under default configuration, it has a a maximum tested range of 10m (can vary depending on the configuration of the DW1000).

[1]Manufactured by the company Bitcraze, based on Malmö, Sweden

3.2 Software

The crazyflie is a big software project, and despite the limitations of working in an embedded system, it provides with modern frameworks for building the code, testing, real time logging and more. The main components are:

- The on-board software: contains all the code run by the quadcopter. It will be covered in detail in the next sections.
- The crazyflie client: written in python, it is in charge of interacting with the quadcopter in real time: shows the console output, allows flashing new firmware, send flight configuration, retrieve information about the status... For development, the most useful characteristics are configuring the logging subsystem and retrieving logging information. It also offers the possibility of saving that information to a file, or plotting it in a graph in real time.

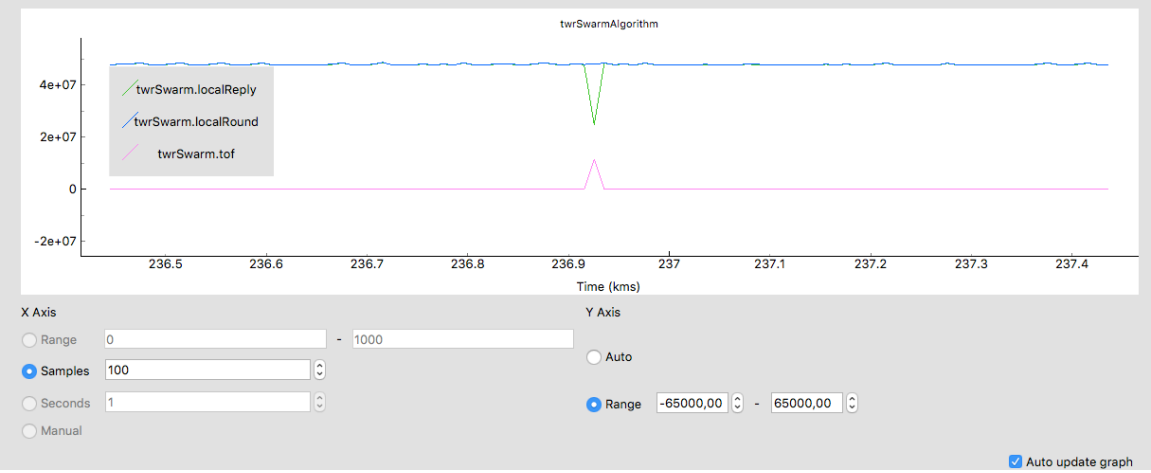

Figure 3.1: The plotter tab, in the crazyflie client

3.2.1 On-board software

The on-board software executed by the crazyflie is divided in different components. This section introduces all of them, and the steps that are performed since the moment the code is written, to its execution on the quadcopter.

3.2.1.1 The building process

The building process groups the steps required to obtain a binary that can be flashed and executed on the quadcopter:

1. The first step happening after writing the code is compilation. Each source file is converted into machine code that contains the processor instructions needed for the

execution of the program. Thus, already at the compilation step, it is necessary to know the architecture of the processor that will be running the program.
In general, when doing software development, the resulting application will be executed in the same machine that performs the compilation. But developing for embedded systems is an exception: because they have limited power and memory, this is not possible. Instead, the compilation happens in a machine with enough resources to perform it, and then the resulting binary is moved to the embedded system, where is executed. This process is known as cross-compilation.

2. After cross-compilation, the next step is linking, which involves taking all the machine code files generated after compilation, and placing them together in a single executable. This executable is organized in sections storing different kinds of data required by the program. In order to give some examples, it is possible to inspect the crazyflie firmware. The command line tool `readelf` can provide us with information about the binary. Among it, the sections organization:

```
There are 26 section headers, starting at offset 0x2b819c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .isr_vector       PROGBITS        08004000 004000 000188 00   A  0   0  1
  [ 2] .flashtext        PROGBITS        08004188 045984 000000 00   W  0   0  1
  [ 3] .text             PROGBITS        08004188 004188 02b1c4 00  AX  0   0  8
  [ 4] .libc             ARM_EXIDX       0802f34c 02f34c 000008 00  AL  3   0  4
  [ 5] .flashpatch       PROGBITS        00000000 045984 000000 00   W  0   0  1
  [ 6] .endflash         PROGBITS        00000000 045984 000000 00   W  0   0  1
  [ 7] .data             PROGBITS        20000000 030000 000d78 00  WA  0   0  8
  [ 8] .bss              NOBITS          20000d78 030d78 014c00 00  WA  0   0  8
  [ 9] .nzds             PROGBITS        20015978 045978 00000c 00  WA  0   0  4
  [10] ._usrstack        NOBITS          20015984 045984 000100 00  WA  0   0  1
  [11] .stabstr          STRTAB          00000000 045984 0001b9 00      0   0  1
  [12] .comment          PROGBITS        00000000 045b3d 00007e 01  MS  0   0  1
  [13] .ARM.attributes   ARM_ATTRIBUTES  00000000 045bbb 000030 00      0   0  1
  [14] .debug_aranges    PROGBITS        00000000 045beb 004a10 00      0   0  1
  [15] .debug_info       PROGBITS        00000000 04a5fb 0bd132 00      0   0  1
  [16] .debug_abbrev     PROGBITS        00000000 10772d 01b33e 00      0   0  1
  [17] .debug_line       PROGBITS        00000000 122a6b 04752f 00      0   0  1
  [18] .debug_frame      PROGBITS        00000000 169f9c 00dd0c 00      0   0  4
  [19] .debug_str        PROGBITS        00000000 177ca8 0afed8 01  MS  0   0  1
  [20] .debug_loc        PROGBITS        00000000 227b80 0342d5 00      0   0  1
  [21] .debug_ranges     PROGBITS        00000000 25be55 006ea8 00      0   0  1
  [22] .debug_macro      PROGBITS        00000000 262cfd 035bd1 00      0   0  1
  [23] .symtab           SYMTAB          00000000 2988d0 016620 10     24 4467  4
  [24] .strtab           STRTAB          00000000 2aeef0 0091a4 00      0   0  1
  [25] .shstrtab         STRTAB          00000000 2b8094 000105 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (purecode), p (processor specific)
```

Code Block 3.1: Sections information extracted from the crazyflie firmware binary by running the command `arm-none-eabi-readelf -S cf2.elf`

Some of the most important sections are:

- `.isr_vector` and `.flashtext`: startup code, specific per platform.
- `.text`: used to store the program's code. Contains read only executable instructions. It also contains read only data: the constants declared in the program. For example, a constant declared as `static const int a = 0;` will be stored in this section.
- `.data`: stores initialized data, which is global and static variables that are assign a value next to their declaration. For example, a static variable declared as `static int a = 0;` will be stored in this section.
- `.bss`: stores uninitialized data, which is global and static variables that do not have a value assigned next to their declaration. For example, a static variable declared as `static int a;` will be stored in this section. The initial value given to this uninitialized static variables depends on the compiler. In case of the crazyflie, which uses the C11 standard (a short name for ISO/IEC 9899:2011) [20], the uninitialized static pointers are automatically initialized to `NULL`, and uninitialized static arithmetic types (integers, floating point numbers...) are initialized to zero. The way this initialization is performed is by zeroing the whole `.bss` section at program startup.
- `.nzds`: this section also stores uninitialized data. The difference with the `.bss` section is that it is not zeroed out at program startup, and thus its value is indeterminate.

The positions of the sections in the final binary are organized by the linker using linker scripts. From [21]: "the main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file". Thus, information about the amount, type and layout of memory is required in this step.
In addition, the linker script configures which sections should be copied to Random Access Memory (RAM) at program startup. For example, the sections `.data` and `.bss` store data that will be read and written during the execution of the program, and have to be located on RAM.

With the information from the link scripts, it is possible to move the binary from the computer used for compilation to the flash memory inside the embedded system, and place the sections in their proper address locations. This process is commonly known as flashing.

3.2.1.2 The software components

As introduced before, the crazyflie has two MCUs: the STM and the NRF. Each one of them requires several binaries in order to be able to boot and run the firmware. And each of those binaries has to go through the building process specified above. The complete memory layout for both chips is as follows:

Data	Start	End	Length (Bytes, hex)	Length (KB)
NRF SoftDevice	0	15FFF	16000	88
NRF Firmware	16000	39FFF	24000	144
NRF Bootloader	3A000	3EFFF	5000	20
NRF MBS	3F000	3FFFF	1000	4
Total				**256**

Table 3.1: Flash memory layout for the NRF

Data	Start	End	Length (Bytes, hex)	Length (KB)
STM Bootloader	8000000	8003FFF	4000	16
STM Firmware	8004000	80FFFFF	FC000	1008
Total				**1024**

Table 3.2: Flash memory layout for the STM

Every binary has its own repository and link scripts:

- NRF SoftDevice [22]: a proprietary binary from the company Nordic Semiconductor, responsible for the Bluetooth communication. It includes two components:
  - The NRF SoftDevice S110 Bluetooth stack, which includes the main code.
  - The NRF Master Boot Record (MBR) section, used to update of the SoftDevice S110 Bluetooth stack if necessary. This is the first binary to be executed after the quadcopter starts.
- NRF Master Boot Switch (MBS) [23]: handles the power button and communicates the duration of the press to the NRF Bootloader. It is also in charge of sending orders to the NRF MBR for flashing the NRF Bootloader or the Softdevice.
- NRF Bootloader [24]: the binary in charge of booting the STM. It is also in charge of flashing the NRF Firmware. There are three different scenarios depending on the duration of the press:
  - Short: starts the crazyflie normally.
  - Long (>3s): starts the crazyflie in bootloader mode, allowing Over the Air (OTA) updates of the NRF and STM firmwares.
  - Very long (>5s): this starts the crazyflie in recovery mode (this mode is in most of the cases never used).
- NRF Firmware [25]: the main application running in the NRF. It is in charge of:
  - Power management.

  - Radio and Bluetooth communication: talks to the Crazyradio PA and with other Bluetooth peripherals.

- STM Bootloader [26]: the binary in charge of updating and flashing the STM firmware.
- STM Firmware [27]: the main application running in the STM.

The goal when splitting the code in different binaries is to enable OTA updates, while making sure that an error during the process does not leave the quadcopter unusable.

3.2.1.3 The booting sequence

The steps performed by the crazyflie for booting the STM Firmware are:

1. The NRF SoftDevice MBR starts. It passes the execution to the NRF MBS.
2. The NRF MBS detects that the power button is pushed shortly. It passes execution to the NRF Bootloader, indicating the length of the push.
3. The NRF Bootloader reads a short push. As a result, it powers up the STM and notifies it that it should start the STM Firmware. Then, it passes the execution to the NRF Firmware.
4. Once the NRF Firmware and the STM Firmware are ready, the quadcopter finished booting.

3.2.1.4 The flashing sequence

The most common way of updating the software in the crazyflie is by flashing the NRF or STM Firmware using the radio. The whole boot sequence for flashing is as follows:

1. The NRF SoftDevice MBR starts. It passes the execution to the NRF MBS.
2. The NRF MBS detects that the power button is pushed for 3 seconds. It passes execution to the NRF Bootloader, indicating the length of the push.
3. The NRF Bootloader reads a medium length push. As a result, it powers up the STM and notifies it that it should start the STM Bootloader. Then, it enables radio and Bluetooth communication.
4. Once the NRF Bootloader and the STM Bootloader are ready, the cload tool (used to send the binary from the development computer to the quadcopter using the Crazyradio PA) can:
   - Communicate to the NRF Bootloader in order to flash the NRF firmware.
   - Communicate to the STM Bootloader in order to flash the STM firmware. Because the radio is managed by the NRF, the NRF acts as a bridge, setting up a transparent communication channel between the client and the STM.

3.2.2 The STM Firmware

The STM Firmware is the main application running on the quadcopter under normal operation: it includes the drivers and the operating system necessary to control, monitor and manipulate data. All this code is part of a standalone project [27]. It is written in C. Its main components are:

- The operating system: the operating system is Free Real Time Operating System (FreeRTOS) V8.2.3, a free, open source, real time operating system written in C, and the "de-facto standard solution for microcontrollers and small microprocessors" [28].
- Deck drivers: the STM Firmware includes the drivers for all the decks manufactured for the quadcopter. They are initialized only if the corresponding deck is mounted on the crazyflie. Under this section is where the code related with the LPS deck is located, together with the implementation of the possible communication schemes (TWR, TDOA).
- The logging subsystem: the crazyflie includes a powerful logging implementation [29]. It allows to log variables from the quadcopter during run time, in an effective way. The logging information is sent to the crazyflie client using the radio. There are two main limitations:
  - Each packet is limited to 32 bytes, which sets a limit on the number of variables that can be logged at the same time: this number depends on the types of the variables, and it will generally be in the order of 10 (for example, if all the values being logged are `float32`, then it will be possible to log up to 8 of them at the same time).
  - The minimum period for logging is 10ms.

The project also provides several tools to assist during development:

- The build system: the STM Firmware is cross-compiled using the GNU Arm Embedded Toolchain [30], which uses *make* [31] together with GCC [32] for building and compiling. Thus, the build system is based on *makefiles*.
- The testing subsystem: the project comes with a testing framework that allows to write unit tests for most of the code. The tests are compiled and run from the development machine, and not from the quadcopter (they are not cross-compiled). It is based on:
  - CMock [33]: a framework that can mock C functions. This is useful when testing a module that calls function in other external modules: it allows to replace their implementation and return known values, so the tests are self contained, independent and executed in a known environment.
  - Unity [34]: a framework that provides the necessary functions to verify the test results, mainly assertions for many different data types.

Unit testing has even more importance in embed systems than in other kind of projects: they typically have none or limited debug capabilities. For example, in order to modify the STM Firmware and run it in the crazyflie, is has to be compiled, flashed to the quadcopter and executed. This usually takes between three to five minutes. After that and during operation, the debug capabilities are limited to logging, and most of the times when something goes wrong the systems reboots without indicating the source of the problem.
Testing the code the way it is done in this project allows to catch bugs and implementation errors almost instantly, because building and running the tests takes usually less than 5 seconds. In addition, writing tests improves code quality and helps to ensure that everything is working as expected when further code modifications occur.

The project does not require any Integrated Development Environment (IDE) (it is even possible to edit the source code with a text editor and build it with make), but using it can speed up the development and make it more pleasant. Some interesting features to take into account for choosing an IDE are syntax highlight, code completion, jump to definition, call the build system from inside the IDE, show errors and warnings next to the lines that generated them, refactoring tools (like search and replace), version control from inside the IDE... Considering those features and after trying several IDEs (Eclipse, CLion), the development is carried out using Xcode. A setup guide can be found in [35], highlighting the necessary configuration and benefits that it provides.

CHAPTER 4
Problem breakdown

This chapter breaks down the the relative positioning problem into several challenges. Its contents are organized as follows:

- Section 4.1 highlights the challenges of designing the ranging scheme.
- Section 4.2 highlights the challenges presented by the development platform in terms of speed and amount of memory available on board of the quadcopter.
- Section 4.3 highlights the challenges presented by the communication channel in terms of several channel characteristics, such as the type, the maximum range or the maximum throughput.

4.1 Challenges when designing the ranging scheme

This section highlight the challenges of designing the ranging scheme. Each one of them presents some questions that guide this master thesis during the next chapters:

1. What data should be exchanged: exchange the minimum amount of data to calculate the TOF, and do it as fast and efficient as possible. The faster the protocol is, the higher amount of ranging information available, and the higher the accuracy that the system can achieve.
   Some questions related to this challenge are:
   - What is the minimum amount of data required for calculating the TOF between two quadcopters?
   - How many transmissions are required to calculate the TOF between two quadcopters?
   - How many transmissions are required to calculate the TOF between all the quadcopters in the swarm?
   - What is the maximum transmission throughput we can get with the available hardware and software?

2. Criteria for starting a transmission: in an scenario where multiple quadcopters have to exchange ranging information using radio signals, each transmission reaches all of them. This may lead to packet collisions, and should be taken into account during the design of the ranging scheme: in order to avoid them, it is necessary that the transmissions happen at different time frames. But, without prior knowledge of how many quadcopters participate in the swarm, deciding when to transmit becomes a complex issue.
   Some questions related to this challenge are:
   - How to coordinate the transmissions between the members of a swarm?

- What is the maximum number of transmissions that can be achieved per second while avoiding collisions?

3. Manage quadcopters when they approach or leave the swarm: the ranging algorithm should be able to incorporate newcomers. Also, it has to detect when a quadcopter stopped responding or left the swarm.
   Some questions related to this challenge are:

   - What is the criteria that defines a swarm?
   - What happens when one of the quadcopters in a swarm stops responding?
   - Would it be possible for an already created swarm to accept new members?

4. Influence of the environment in the wireless transmission: from the moment the radio signals leave the transmitting antenna until the point they reach their destination, there are two main phenomenons that may affect the transmission:

   - Multipath propagation: radio transmission can experience that a packet sent once is reaching the destination antenna multiple times. This happens mainly due to refraction and reflection of the waves, which bounce into walls, floor or other obstacles on the way:

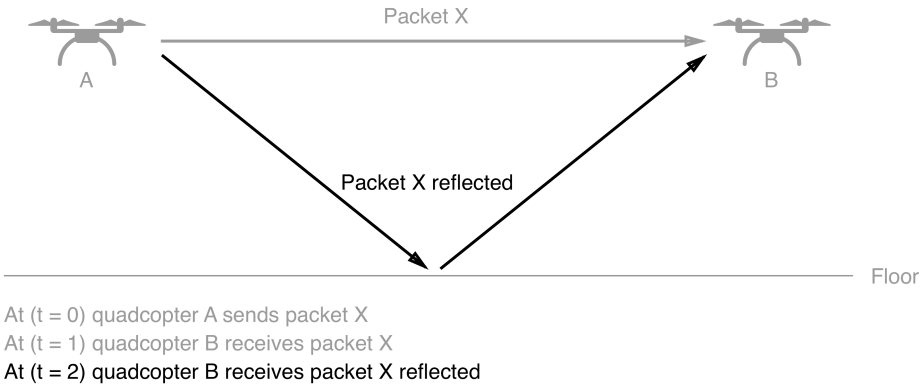

Figure 4.1: A reflected radio packet reaching the antenna

   - Signal interference: radio signals may be modified during transmission, due to interference from other wireless systems, or due to packet collisions. The result is that the sent packets may never reach their destinations, or if they do, they can be malformed.

   Some questions related to this challenge are:

   - What happens when a collision takes place? Will the packets reach their destination?
   - Is it possible to detect collisions?
   - Is it possible to detect when multipath propagation occurs?

4.2 Challenges introduced by the development platform

The implementation and performance of the ranging scheme may be constraint by the specifications of the STM (used as main controller) or the DW1000 module (in charge of the UWB communications):

- The clock frequency of the STM, 168MHz, could be a limiting factor of the precision of the final system: if it is not able to process the incoming packets fast enough, the maximum throughput of the system may be reduced.
- The available memory, 192kb of RAM and 1Mb of flash, is a usual limitation in embedded systems. Traditional computers use powerful operating systems that apply different techniques (like virtual memory or swapping) to emulate an scenario where the available memory for the applications is infinite. But in the case of embed systems, where the operating system is way smaller and simple, the physical installed memory is the only memory available for the applications. When this memory is filled or closed to be filled the system may enter in an unpredictable state, which in case of a quadcopter generally means that will stop operating and fall to the ground.
  Also, the way this memory is used present some challenges. There are two ways in which an object may be allocated in memory [36, 37]:
  - Statically: the amount of memory required to run the application is known at compile time. This means that the size of the data that the application requires for functioning needs also to be known at compile time. In many cases this is not possible and leads to the developer reserving more memory than what is really needed, which is a problem in systems with limited amounts of memory (like embed systems). Also, knowing how the memory is allocated at compile time makes it faster when using it at run time, because it knows in advance where the objects should be allocated in memory. Finally, using static memory does not require any management: there is no need to write additional code for allocating or freeing the memory when using it because the system is able to do it automatically.
  - Dynamically: allocates memory at run time, which does not require to know the size in advance. This allows using exactly the required memory, avoiding memory waste. the fact that this allocation is performed at run time requires the system to look for a free area in memory where the object to be allocated fits, and thus is slower than static allocation. Also, it is necessary to keep track and manage the usage of memory: the developer controls the life time of the allocated objects, which means that the memory needs to be manually freed if not needed anymore (otherwise, after multiple allocations the system would suffer a heap overflow).
- The clock frequency tolerance: in order to generate the frequency at which a MCU operates, a hardware component called oscillator is required. During its production phase, the oscillation period is adjusted, which allows the manufacturer to specify the generated frequency as a central value and a possible maximum deviation (also called error or tolerance). This means that every chip manufactured has a slightly

different frequency. Because in this project UWB is used for distance calculations and this radio signals travel at the speed of light, the frequency tolerance can have an important impact on the resulting measurements.

4.3 Challenges introduced by the communication channel

For transmission and communication between quadcopters, The DW1000 UWB radio transceiver is used [38, 39]. There are some characteristics of this chip important to highlight:

- Provides a half-duplex communication channel, which means that it can not transmit and receive at the same time.
- The maximum range according to the manufacturer is 290 meters. The real usable range is expected to be lower and depends on the configuration of the chip.
- The maximum data throughput provided by the manufacturer is 6.8Mbps. The real value while operating depends on the configuration of the chip.

In order to get an idea of the real values for maximum throughput and range, two simple tests have been done. The results are as follows:

- Maximum throughput test: the measurements can be seen in appendix A. They show that the maximum throughput we can expect from the system with the default configuration and sending empty packets, is of approximately 1530 successful ranging per second, which is approximately 3000 packets per second.
- Maximum range test: performed with two quadcopters using the default configuration, the results show that the maximum range is around 10 meters, in straight line and without obstacles.

CHAPTER 5
Design

This chapter introduces the design of the system. Its contents are organized as follows:

- Section 5.1 analyses the trilateration problem depending on the number of dimensions. It serves as an introduction to the next sections.
- Section 5.2 explains the ranging scheme used.
- Section 5.3 explains what additional information should be transmitted in the packets in order to be able to calculate the position of each member of the swarm.

5.1 Trilateration: N-Dimension analysis

The position calculation is based on trilateration: given multiple known points P_1, P_2, P_3... P_M and knowing the distances d_{t-1}, d_{t-2}, d_{t-3}... d_{t-M} between them and an unknown point P_t, obtain the coordinates of P_t.
For now on, the point to be calculated P_t will be named tag, and the reference points $P_1, P_2, P_3...P_M$ will be called neighbours.

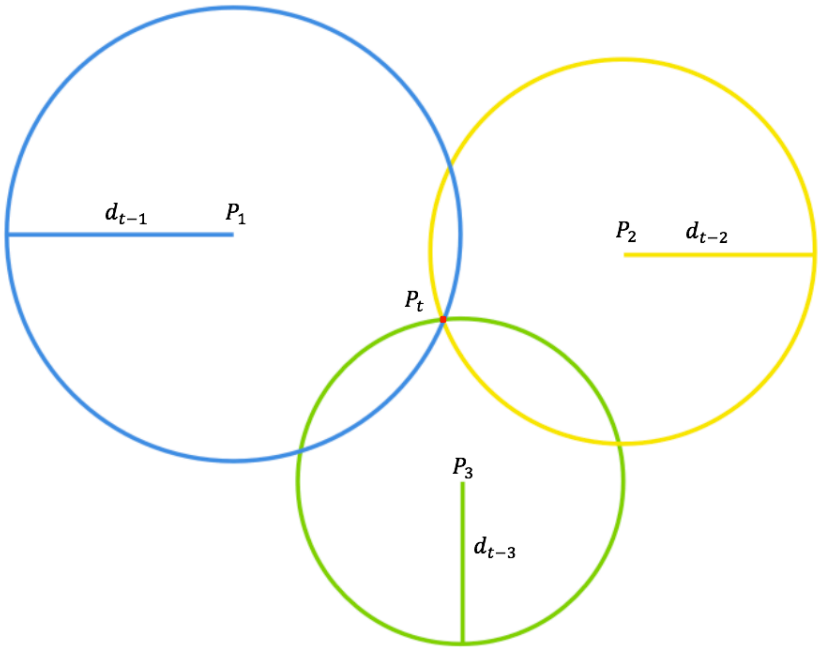

Figure 5.1: An example of trilateration in 2D

But, what are the minimum number of neighbours that are required for obtaining P_t in two and three dimensions?

Consider a N-dimensional space. The relative position of the tag respect to one neighbour can be express as:

$$R = f(P_i, d_{t-i}),\ where\ P_t \in R \tag{5.1}$$

There are multiple solutions for this equation (R), and the amount depends on the number of dimensions of the space. For example:

- In a 1D space, there are two solutions: two points, that together with P_i form a line, and that both of them are at the same distance d_{t-i} from P_i.
- In a 2D dimensional space, the solutions form a circumference, with center P_i and radius d_{t-i}.
- In a 3D dimensional space, the solutions form an sphere, with center P_i and radius d_{t-i}.

Having M neighbours and M distances, there are M equations like the equation 5.1 to calculate P_t, which result in multiple R_i solutions. Calculating the intersection between each R_i reduces the number of possible results. Concretely, every intersection reduces the N-dimensional space where the solutions exist by one:

- In a 1D dimensional space, having two known points and the distances between them and the tag, and calculating their intersection, leaves only one possible result for P_t.
- In a 2D dimensional space, having two known points and the distances between them and the tag results in two circumferences. Calculating their intersection, leaves us with a solution that exist in a 1D dimensional space: two points.
- In a 3D dimensional space, having two known points and the distances between them and the tag results in two spheres. Calculating their intersection, leaves us with a solution that exist in a 2D dimensional space: a circumference.

If we want to reduce the possible solutions for P_t to one (we want to know the exact position, which is a single point), a N-dimensional space requires N intersections of $N+1$ groups of solutions. Thus, the following equation:

$$[P_t]_{N-dimensions} = f(P_1, d_{t-1}) \cap f(P_2, d_{t-2}) ... \cap f(P_M, d_{t-M}),\ where\ M \geq N+1 \tag{5.2}$$

Which means that in a N-dimensional space, for calculating the position of a tag P_t, the position of $N+1$ neighbours P_1,P_2,P_3...P_{N+1} and the distances between each of them and the tag d_{t-1},d_{t-2},d_{t-3}...$d_{t-(N+1)}$ have to be known.

- In a 1D dimensional space, two neighbours and the distances to them are needed:

$$[P_t]_{1D} = f(P_1, d_{t-1}) \cap f(P_2, d_{t-2}) \tag{5.3}$$

- In a 2D dimensional space, three neighbours and the distances to them are needed:

$$[P_t]_{2D} = f(P_1, d_{t-1}) \cap f(P_2, d_{t-2}) \cap f(P_3, d_{t-3}) \tag{5.4}$$

- In a 3D dimensional space, four neighbours and the distances to them are needed:

$$[P_t]_{3D} = f(P_1, d_{t-1}) \cap f(P_2, d_{t-2}) \cap f(P_3, d_{t-3}) \cap f(P_4, d_{t-4}) \tag{5.5}$$

5.2 Ranging scheme

This chapter introduced the theory behind the ranging scheme used in this master thesis. The ranging scheme specifies what information is transmitted between quadcopters, and when is this transmission happening.

5.2.1 Existing ranging schemes

This section introduces existing ranging schemes that are used as a reference in the design of the final ranging scheme.

5.2.1.1 One to one ranging: TWR

The simplest ranging scheme is Two-Way Ranging (TWR). It is specified in the standard IEEE 802.15.4-2011 [40]. The following figure illustrates it:

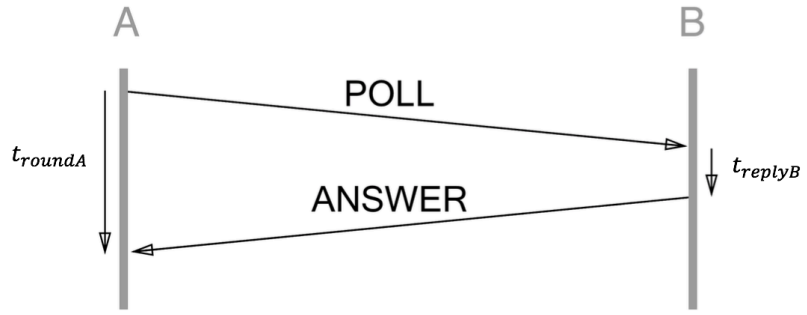

Figure 5.2: The packet exchange during the TWR ranging scheme

Ideally, the calculation of the TOF is obtained as:

$$[tof_{A-B}]_A = \frac{[t_{roundA}]_A - [t_{replyB}]_B}{2} \tag{5.6}$$

Where:

- $[tof_{A-B}]_A$ is the TOF between A and B, measured by the clock of A.
- $[t_{roundA}]_A$ is the time passed between A sends a $POLL$ packet and receives an $ANSWER$ packet, measured in the clock of A.
- $[t_{replyB}]_B$ is the time passed between B receives a $POLL$ packet and sends an $ANSWER$ packet, measured in the clock of B.

The $[t_{replyB}]_B$ information is included by B in the $ANSWER$ packet.

The main issue of this simple ranging scheme is that it assumes that the clocks of A and B run at the same exact frequency. In reality this is not true, and in order to

be able to calculate $[tof]_A$ it is required to obtain t_{replyB} measured by the clock in A: $[t_{replyB}]_A$. Thus, the correct ranging equation that takes this into account is:

$$[tof_{A-B}]_A = \frac{[t_{roundA}]_A - [t_{replyB}]_A}{2} \tag{5.7}$$

Where $[t_{replyB}]_A$ can be calculated as:

$$[t_{replyB}]_A = [t_{replyB}]_B \times e_{A-B} \tag{5.8}$$

$$e_{A-B} = \frac{f_A}{f_B} = \frac{[t]_A}{[t]_B} \tag{5.9}$$

Where:

- t is any arbitrary period of time.
- $[t]_A$ is t measured by the clock of A.
- $[t]_B$ is t measured by the clock of B.

In order to obtain e_{A-B}, the ranging scheme has to be changed.

5.2.1.2 One to one ranging: SDS-TWR

The SDS-TWR scheme includes two more packets exchanges called *FINAL* and *REPORT*. This allows A and B to measure the time between the *POLL* and the *FINAL* packets ($t_{pollToFinal}$), obtaining $[t_{pollToFinalA}]_A$ and $[t_{pollToFinalB}]_B$, which makes possible to calculate the frequency difference e_{A-B}. This assumes that, during the ranging attempt, A and B are at the same distance: $t_{pollToFinalA}$ is equal to $t_{pollToFinalB}$. This is a reasonable assumption because the packets travel at the speed of light, and a ranging attempt is completed in a very short period of time. The following figure illustrates it:

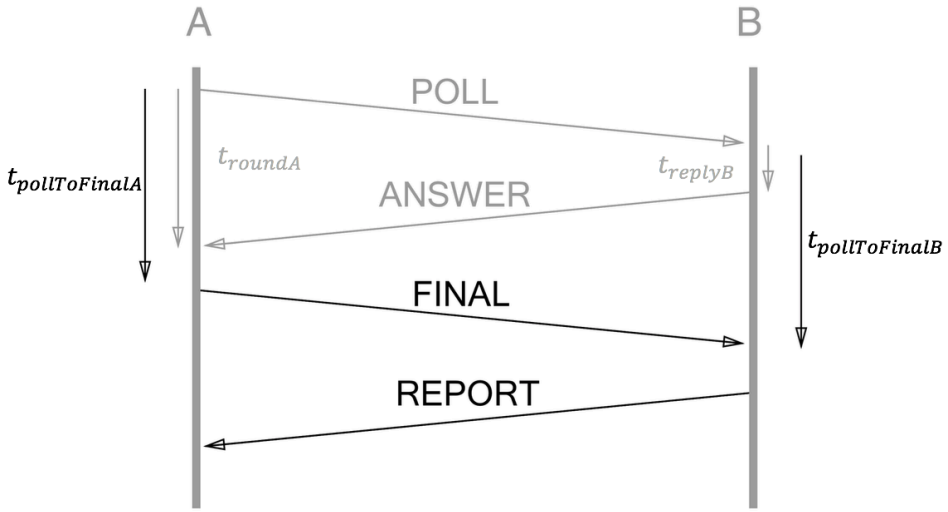

Figure 5.3: The packet exchange during the SDS-TWR ranging scheme

Then, B includes $[t_{pollToFinalB}]_B$ in the *REPORT* packet. After it, A has all the necessary information for calculating TOF. From equation 5.7:

$$[tof_{A-B}]_A = \frac{[t_{roundA}]_A - ([t_{replyB}]_B \times \frac{[t_{pollToFinalA}]_A}{[t_{pollToFinalB}]_B})}{2} \tag{5.10}$$

Where:

- $t_{pollToFinal}$ is the time between the *POLL* and the *FINAL* packets.
- $[t_{pollToFinalA}]_A$ is $t_{pollToFinal}$ observed in A and measured by the A clock.
- $[t_{pollToFinalB}]_B$ is $t_{pollToFinal}$ observed in B and measured by the B clock.

5.2.2 Many to many: swarm ranging scheme

In order to adapt the SDS-TWR ranging method to a swarm of quadcopters, the *POLL*, *ANSWER*, *FINAL* and *REPORT* packets are merged: the information traveling in each one of them is now traveling together in one, which will be called *STATUS* packet. The *STATUS* packet includes the following information:

- $[t_{replyXtoM}]_X, for M = 1, 2...N-1$, being N the total number of quadcopters in the swarm, and X the quadcopter sending the *STATUS* packet. $[t_{replyXtoM}]_X$ Is the time passed since the last time that X received a packet from M, measured by X.
- $[t_{pollToFinalX}]_X$ is renamed to $[t_{sinceLastPacketX}]_X$, being $[t_{sinceLastPacketX}]_X$ the time passed since X last sent a *STATUS* packet.

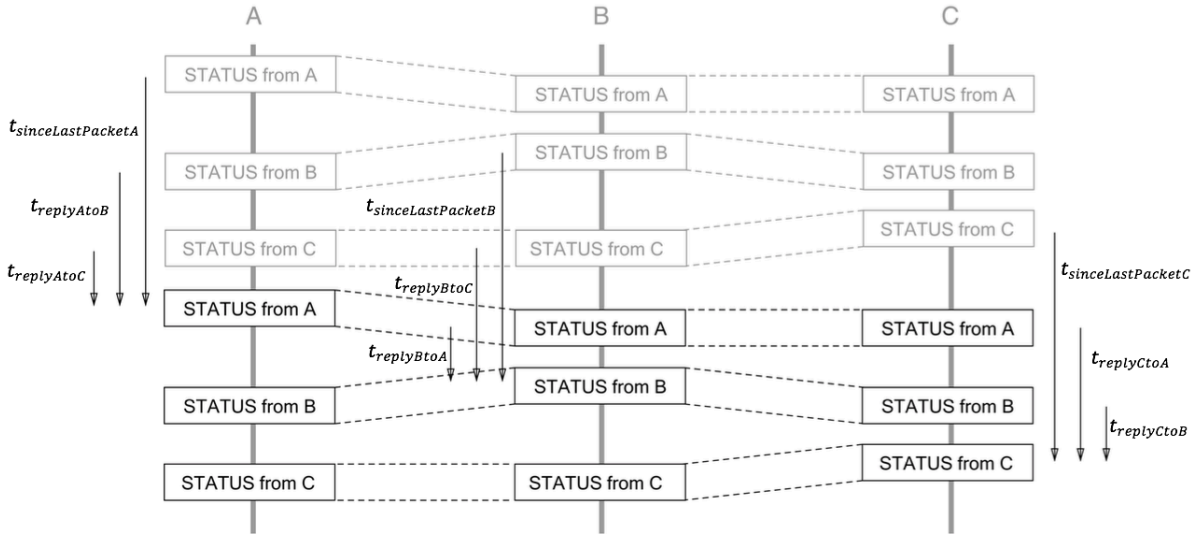

Figure 5.4: The packet exchange during the swarm ranging scheme

With this information, a quadcopter M receiving the *STATUS* packet from his neighbour X can use the equation 5.10 to calculate the TOF:

$$[tof_{M-X}]_M = \frac{[t_{roundMtoX}]_M - ([t_{replyXtoM}]_X \times \frac{[t_{sinceLastPacketX}]_M}{[t_{sinceLastPacketX}]_X})}{2} \tag{5.11}$$

Where:

- $[t_{roundMtoX}]_M$ is the time between M sends its *STATUS* packet and receives the *STATUS* packet from X, measured by M.
- $[t_{replyXtoM}]_X$ is the time between X receives the *STATUS* packet from M and sends its *STATUS* packet, measured by X.
- $[t_{sinceLastPacketX}]_M$ is the time between two packets arriving to M from X, measured by M.
- $[t_{sinceLastPacketX}]_X$ is the time between two packets are sent from X, measured by X.

5.3 Position estimation

This chapter explains what additional information is required on top of the timestamps introduced in section 5.2.2 to estimate the position of a quadcopter relative to the position of its neighbours:

- The TOF values calculated by each individual quadcopter need to be broadcast to all of them.
- In order to obtain a coordinate to express the relative position of the quadcopters in the swarm, each quadcopter should establish its own coordinate system.
- In order to be able to have a stable position the problem of position drifting should be taken into consideration.

5.3.1 Broadcast of TOF values

The ranging scheme introduced in section 5.2.2 is used to calculate the TOF between two quadcopters. Thus, after the whole swarm completes a ranging cycle the quadcopters know the TOF between themselves and their neighbours, but not the TOF between neighbours. The following figure illustrates this:

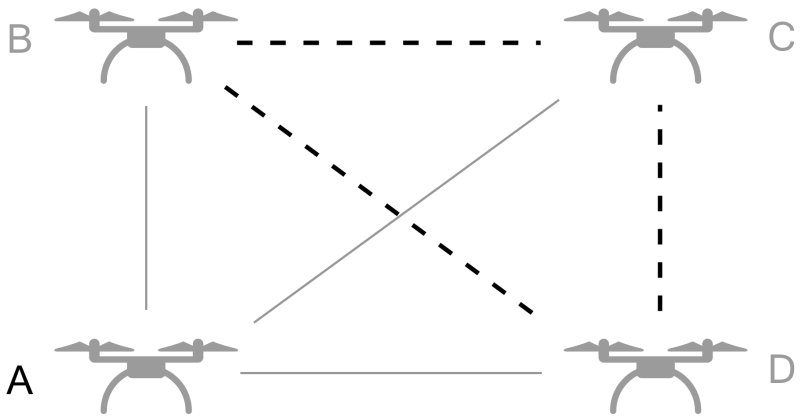

Figure 5.5: The TOF values known/unknown by quadcopter A after a full ranging cycle, in full/dotted lines

In order to solve this, the quadcopters have to add the TOF between themselves and their neighbours in their *STATUS* packet. The following figure illustrates this:

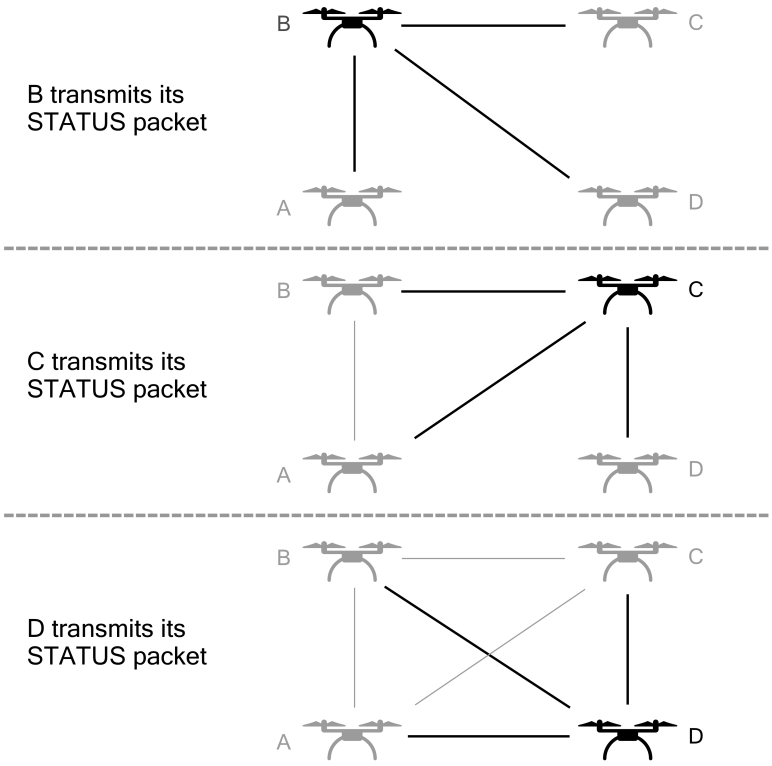

Figure 5.6: The TOF values known by quadcopter A as the rest of the neighbours transmit their *STATUS* packets. In black, the TOF values resulting from the quadcopter transmitting. In gray the TOF values resulting from previous quadcopters transmitting

After a whole ranging cycle, all quadcopters know the TOF value between any two quadcopters in the swarm. With this information and using equation 5.5, quadcopters are able to trilaterate their position relative to the position of their neighbours.

5.3.2 Generate a coordinate system

When all the TOF values between all quadcopters in the swarm are known, it is the moment of trilaterating the relative position of all of them. But, in order to do this, having the distance (TOF) between them is not enough: a coordinate system is required. In order to establish it and obtain its origin and orientation, several assumptions are needed:

1. The origin of the coordinate will be established by the first detected neighbour: $P_1 = (0, 0, 0)$.
2. The orientation of the coordinate system will be given by the next 3 detected neighbours (assuming a 3D space):

- The second quadcopter will define the X axis and its direction: $P_2 = (d, 0, 0)$, where d is the distance between the first and second quadcopter.
- The third quadcopter will define the XY plane, and the direction of the Y axis: $P_3 = (dx, dy, 0)$. The coordinates (dx, dy) is the 2D result of trilaterating P_3 position from P_1 and P_2, and from the two possible solutions choose the one with $dy > 0$.
- The fourth quadcopter will define the direction of the Z axis: $P_3 = (dx, dy, dz)$. The coordinates (dx, dy, dz) is the 3D result of trilaterating P_4 position from P_1, P_2 and P_3, and from the two possible solutions choose the one with $dz > 0$.

After the fourth detected neighbour, the position of each quadcopter will be trilaterated based on the position of all the known neighbours, which will be based on the created coordinate system. The following figure illustrates the process:

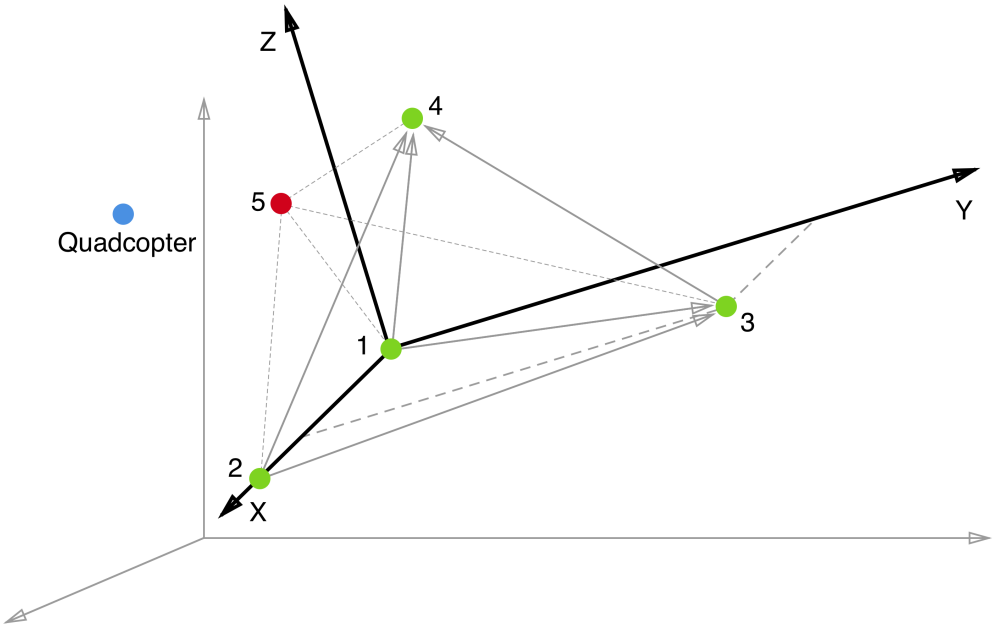

Figure 5.7: Coordinate system creation. In blue, the quadcopter. In green, the neighbours building the coordinate system. In red, a neighbour positioned based on the already built coordinate system

5.3.3 Position drifting

In a relative positioning system, changing the position of the reference neighbours results in a change of the position of the quadcopter, even if the quadcopter is not moving.

In an ideal setup, the TOF from one quadcopter to the neighbours around will result always in the same value. In reality, the measurement of the TOF includes error. This error modifies slightly the calculated position of the quadcopter, which will then be used

as a reference to calculate the position of other quadcopters. As a result, a small error in one TOF measurement can lead to position changes in all the quadcopters of the swarm. The following figure illustrates this problem: it shows two quadcopters calculating their position relative to each other. One of them always calculates a slightly longer TOF, resulting in drifting of the whole swarm over time:

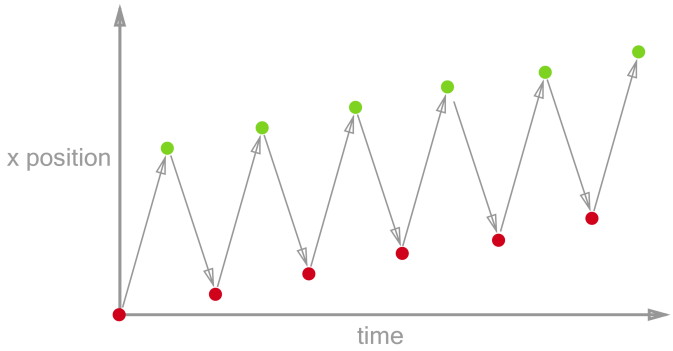

Figure 5.8: 1D relative positioning of two stationary quadcopters, where one of them calculates slightly longer TOF than the other

In order to overcome this issue, the positioning system needs to implement logic to be able to reject position changes that are due to TOF measurement errors, and only accept the ones that happen due to legitimate position changes.

CHAPTER 6
Implementation

This chapter introduces the development process. Its contents are organized as follows:

- Sections 6.1 to 6.7 explains the implementation details of the swarm ranging scheme introduced in section 5.2.2.
- Section 6.8 explains the implementation details of the position calculation introduced in section 5.3.
- Sections 6.9 to 6.10 explains several problems found during development, their origin and the applied solution.

6.1 Packet identification

When operating, the DW1000 antenna captures any received packets, and uses their content assuming that they are of the *STATUS* type. This works when all the quadcopters around are using the swarm ranging scheme, but the application malfunctions in case any of the quadcopters is using the DW1000 for a different application (and the packets have different contents). For example, it may be that in the same room there is a swarm of quadcopters implementing the swarm ranging scheme, and some anchors implementing TDOA.

In order to overcome this problem it is necessary to identify which of the packets are *STATUS* packets, and discard the rest. For this purpose a packet identifier is also transmitted, which is common and unique to all *STATUS* packets. All incoming packets are checked against the identifier, and discarded if not matching.

6.2 Quadcopter identification

In order to be able to relate the data coming in the packets with the quadcopters it is necessary to assign them an identifier. The complexity of this task is to ensure that they are unique inside a swarm without knowing in advance the number of neighbours.

The approach followed is to generate the identifier using the `int rand(void)` function, a pseudorandom number generator. The main caveat is that the sequence of values it returns is predictable if its first value is known. It requires a starting seed that should be different every time. The common practise is to use the system clock as a seed.

During the implementation of this feature, using the system clock generates the same identifiers almost all the time. This is due to the fact that the clock information provided by FreeRTOS has a precision of milliseconds: after booting the crazyflie, the seed is set during the same millisecond in most of the cases.

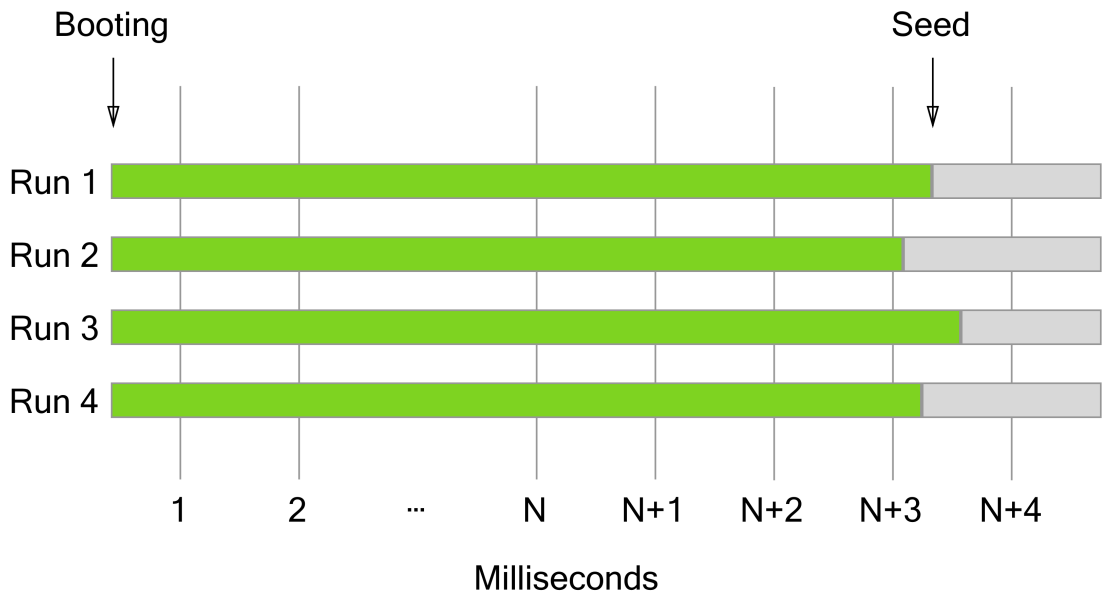

Figure 6.1: Representation of four times the crazyflie was booted, where the seed was set in the same millisecond

In order to generate a random seed, a clock with higher frequency is needed. The DW1000 clock can be used for this purpose[1]. After several tests, its higher frequency proved to generate different seeds most of the times, leading to pseudorandom identifiers that are random enough for this application.

The size of the identifier should be:

[1]Refer to section 6.4.1 for more information about its frequency

- Big enough, so the probabilities of generating two random identifiers that are equal is low.
- As small as possible, to reduce the size of the data travelling in the packets.

The chosen size is `uint8_t`, which allows identifiers form 0 to 255.

But still, with such a random generation of identifiers it is possible that two quadcopters in a swarm end up with the same identifier. To mitigate this, the following functionality has been implemented:

- Detection of a quadcopter with the same identifier: when a packet arrives, the id of the transmitting quadcopter is compared against the receiving quadcopter.
- Regeneration of the identifier: when a match is found, a new identifier is generated, making sure that it is different from:
  - The identifier of the transmitting quadcopter, located in the packet.
  - The identifiers of the neighbours of the transmitting quadcopter, located in the packet.
  - The identifiers of the neighbours of the receiving quadcopter, located in the memory of the receiving quadcopter.

As a result, the following API controls the identifier generation:

```
void initRandomizationEngine(dwDevice_t *dev);
locoId_t generateId(void);
locoId_t generateIdNotInPacket(neighbourData_t neighboursStorage[],
↪  lpsSwarmPacket_t* packet);
```

Code Block 6.1: API for generating quadcopter identifiers

- `initRandomizationEngine`: seeds the `int rand()` function using the DW1000 clock. Called once during algorithm initialization.
- `generateId`: uses the `int rand()` function to generate a random identifier. Called once during algorithm initialization.
- `generateIdNotInPacket`: regenerates the identifier, making sure that is not an identifier already in use. Called any time a duplicated identifier is detected.

6.3 Preventing duplicated packets: the sequence number

This section explains the introduction of sequence numbers on the packets in order to avoid the multipath propagation problem explained in section 4.1.

This implementation is focused on discarding duplicated packets that arrive to the antenna. The sequence number has a size of 1 byte, wrapping around every time it reaches the its maximum value. It rejects packets with a sequence number between one and five units lower than the expected one. Thus, when a duplicated packet arrives just after the original one, both of them have the same sequence number, but only the first will be accepted.

The storage requirements are:

- Each quadcopter has to keep a variable to set the sequence number of its transmitted *STATUS* packets.
- Each quadcopter has to keep one variable per neighbour in order to track their sequence numbers.

The implementation holds this variables in the `ctx` and `neighbourData_t` structs[2].

[2]Refer to appendix D for the source code

6.4 Timestamping tx and rx events: DW1000

This section talks about the implementation details of generating the timestamps for transmission and reception events.

As shown in 5.2.2, in order to be able to calculate the TOF it is necessary to know when a transmission or reception happens. The DW1000 is used for this purpose. Its main characteristics are:

- Its timestamp frequency: the standard IEEE 802.15.4-2011 [40] defines a frequency of operation of 499.2MHz for UWB transceivers. The DW1000 captures the receive and transmit timestamps at a higher frequency of 499.2MHz * 128 $\approx$ 64GHz, so it can provide a precision of 15.65ps.
- The timestamps are stored on a 40 bits register.

The following sections are organized as follows:

- Section 6.4.1 explains the challenges of working with the DW1000, and its hardware limitations.
- Section 6.4.2 explains the implemented solutions.

6.4.1 DW1000 limitations

In order to use this chip for TOF calculations there are some implementation details that should be considered:

- The frequency tolerance: in this application the frequency tolerance of the DW1000 module can reduce the precision of the system. In its datasheet [41], the manufacturer specifies that it has a crystal oscillator reference frequency of 38.4MHz, with a typical tolerance of $\pm$25ppm. This is the error that can be expected after the factory adjustment.
  In addition, the chip has a built-in dynamic trimming system that adjusts the clock frequency during operation, which the manufacturer claims that would reduce the typical error to $\pm$2ppm.
  Because the 64GHz frequency that creates the timestamps is generated from the 38.4MHz oscillator, the tolerance of the oscillator introduces error in the TOF measurements. The manufacturer provides a table specifying this error for different scenarios (table 62 of the DW1000 User Manual [38]). For example, for a frequency deviation of $\pm$2ppm, the TOF error ranges from 0.1ns to 5ns.
  Comparing the precision of the timestamps (15.65ps) and the error introduced by the clock deviation (0.1ns minimum) it is clear that, in order to get accurate TOF measurements, the clock deviation needs to be taken into account.
- Timestamp wrap around: as a result of the high frequency used for timestamp creation and the limited size of the register that stores it, the count will wrap around. The frequency with which this will happen can be calculated as follows:

$$t_{wrap} = \frac{2^{40} - 1}{499.2MHz * 128} \approx 17.21 \ seconds \tag{6.1}$$

The following table show an example of the problems that wrap around can introduced. Considering an scenario where a quadcopter has to calculate its reply time, it requires to know the receive and transmit timestamps. Assuming a high timestamp value for the receive event and the reply time, it is possible to calculate the transmission time:

Rx timestamp	$2^{40} - 800$
Reply time	1300
Tx timestamp	$2^{40} + 500$
Tx timestamp after wrap around	**500**
Calculated reply time	$500 - (2^{40} - 800) = 1300 - 2^{40}$

Table 6.1: Example calculations when the timestamp counter wraps around

Ass shown on table 6.1, the effect of the counter wrapping around gives a negative value for the calculated reply time.

- Antenna delay: as explained in [42], there is a delay between the moment in which a packet reaches the antenna and the moment when it is timestamped.

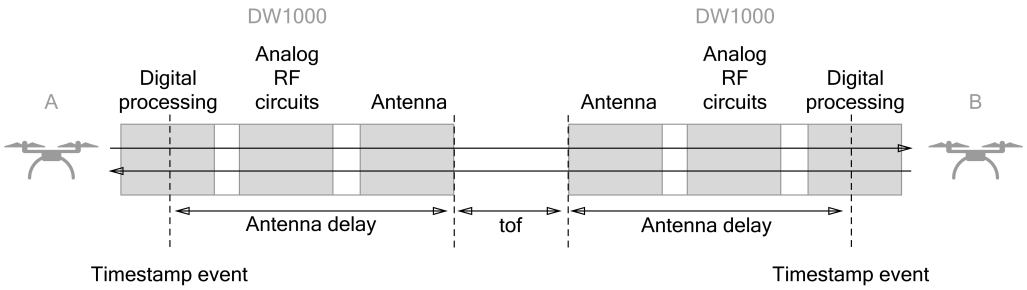

Figure 6.2: Antenna delay diagram

This should be addressed during TOF calculation.

6.4.2 Implemented solutions

In order to be able to do a precise measurement of the TOF, the limitations of the DW1000 should be mitigated. The implemented solutions are:

- Frequency tolerance: as explained in section 5.2.2, the ranging scheme provides the necessary information to calculate the difference in clock frequency between quadcopters. The smaller this difference is, the better the accuracy of the calculated TOF.

In order to reduce it as much as possible, the DW1000 is configured as the manufacturer proposes, using the built-in dynamic trimming system to reduce the typical clock deviation from ±25ppm to ±2ppm. The pull request in [43] includes the necessary changes to enable this feature, and shows the difference between before and after the implementation (even though the pull request was not merged to the main repository [44] because there was a similar implementation done by the maintainers).

Another source of error for the clock correction values is the error introduced by the UWB measurements, which can sometimes lead to completely unreliable results. The following figure illustrates this issue:

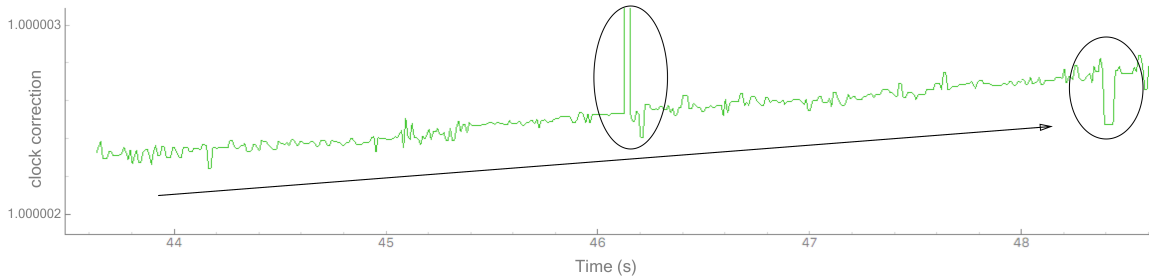

Figure 6.3: Calculated clock correction between two quadcopters

It is possible to observe that:

- The calculated clock correction is steadily growing over time. This is due to the changing temperature: the operation of the quadcopter produces heat and the chips get warmer after they are turned on. After some time the temperature stabilizes and the clock correction stops growing.
- Some of the clock correction calculated values introduce an important error (highlighted with a circle in the figure), compared to the average value.

The clock frequency of a quadcopter is stable and only changes slowly with temperature over time. Knowing this, it is possible to filter the calculated clock correction in order to discard wrong values.

The applied filtering works as follows:

- An initial filtering establish absolute top and bottom limits for the clock correction. Measurements not falling in this range are considered invalid and discarded. Such limits are based on the typical clock deviation of ±2ppm:
  - The top limit has a value of: $1 + 2 * clockDeviation = 1 + 4 * 10^{-6}$
  - The bottom limit has a value of: $1 - 2 * clockDeviation = 1 - 4 * 10^{-6}$
- A second filtering establish an accepted deviation for the clock correction candidate relative to the last calculated clock correction. If the candidate value falls outside

of it, it is considered invalid and discarded. The value is obtained experimentally, and is $\pm 0.03 * 10^{-6}$.

– A candidate value that pass the previous two filters is averaged with the already calculated value of the clock correction.

The following figure shows the result of such implementation:

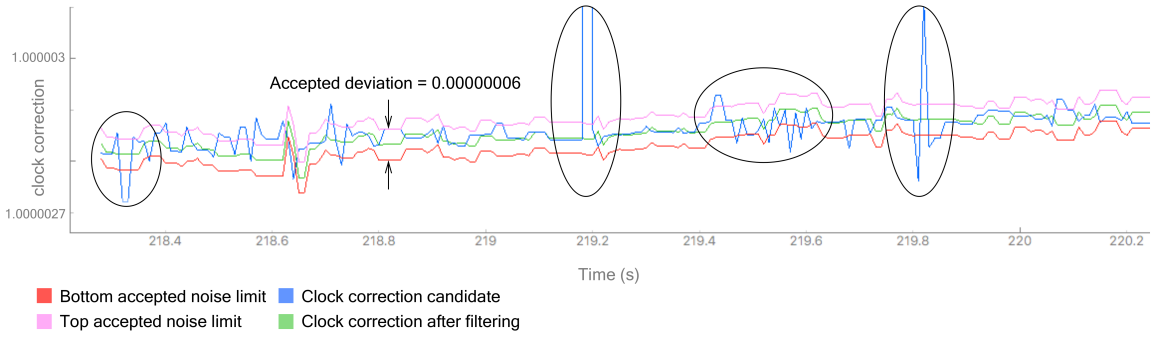

Figure 6.4: Filtered clock correction between two quadcopters. The absolute top and bottom limits are out of the range of the vertical axis

The figure highlights specific moments where the clock correction candidate heavily deviates from the average (blue), but thanks to the filtering the final value is not affected (green).

- Timestamp wrap around: the wrap around problem explained before is the result of operating with signed integer values. The issue can be overcome by using unsigned integer values for the calculations: this unsigned types can not represent negative values, and thus, when the result of an operation is negative, they will represent the equivalent positive value for the binary they store. It is also required to truncate the result bit width to a width equal or smaller than the one used by the counter. Doing the operations with this two requirements has the side effect of avoiding the problems introduced by the DW1000 counter wrap around, giving the correct results in any of the cases.
  To illustrate this concept, the following lines show an example. In order to make it easier to understand, table 6.2 uses the same data and calculations from table 6.1, but using 8 bits to store the values instead of 40:

Rx timestamp	$2^8 - 10 = 246$
Reply time	20
Tx timestamp	266
Tx timestamp after wrap around	10
Calculated reply time (signed)	$10 - 246 = -236 = 100010100b$ (9 *bits*)
Calculated reply time (unsigned)	$00010100b$ (8 *bits*) $= 20$

Table 6.2: Wrap around when using unsigned integers for calculations

Ass seen on the table, taking the binary representation of the negative result and truncating it, gives the correct reply time.
Because the DW1000 counter bit width is 40 bits, the bit width used to make the TOF calculations is 32 (the type used is `uint32_t`). This will throw correct results as far as the time to measure is less than:

$$t_{wrap} = \frac{2^{32} - 1}{499.2MHz * 128} \approx 0.0672 \ seconds \tag{6.2}$$

Thus, it is important to keep a ranging frequency that guarantees that round and reply times are smaller than 0.0672 seconds[3].

- Antenna delay: considering the delay introduced by the antenna, the TOF results as follows:

$$measuredTof_{A-B} = tof_{A-B} + delay_{antennaA} + delay_{antennaB} \tag{6.3}$$

Where $delay_{antennaA} + delay_{antennaB} \approx 515.7ns$, which in the end introduces an error of ≈154.6m in the final calculated distance (assuming signals that travel at the speed of light).

The antenna delay changes slightly for each DW1000 module, and also with temperature. The DW1000 manufacturer proposes to overcome this tolerances by calibrating each DW1000 individually: this solution is not optimal when the calibration has to be performed in multiple quadcopters.
No alternative method has been found in order to get a better measurement of the antenna delay. Thus, an arbitrary measured value of ≈515.7ns is used. This may introduce important errors during position estimation.
In order to store and use the antenna delay value, the DW1000 module offers dedicated memory and functions for it, which returns the real TOF value if configured properly: the DW1000 takes the responsibility of subtracting the antenna delay value from every timestamp calculation. This functionality is not used in the crazyflie firmware: instead, the manipulation of the measured TOF and the corresponding antenna delay adjustment is done at user application level. For example, in this master thesis, the antenna delay is subtracted from the TOF when the *STATUS* packet is received.

[3]Refer to section 6.5 for final calculations to fulfill this requirement

6.5 Randomized transmission

The ranging scheme introduced in section 5.2.2 does not specify the order in which the quadcopters transmit their *STATUS* packets, and leaves that as an implementation decision.

When thinking about the implementation for the first time, the immediate solution is to make the quadcopters transmit in a predefined order, transmitting their *STATUS* packet as soon as they receive the *STATUS* packet from the quadcopter transmitting before them. This scheme is illustrated in figure 6.6. The benefits and drawbacks from this implementation are:

- It avoids packet collisions.
- The transmission time of the first quadcopter (t_{txA}) does not require to be sent in the *STATUS* packet, reducing its size.
- It requires to design and implement logic to configure the order of the quadcopters before the ranging starts.
- Loosing a packet will interrupt the ranging cycle. It requires to design and implement code to manage such scenario.
- It requires to manage the scenario where a new quadcopter joins the swarm, and also when it leaves it.

Such an implementation allows to reduce the size of the packets and reach the highest ranging frequencies, but present important consensus problems.

A second alternative is to randomize the transmission of the data, assuming that collisions may happen. This idea comes from the Advocates of Linux Open-source Hawaii Association (ALOHA) protocol [45] (a random-access protocol developed in the 60s that has been extensively researched). When randomizing the transmissions, an average transmission frequency is maintained, but the transmission events do not keep a fixed period between them. The following figure illustrates this implementation:

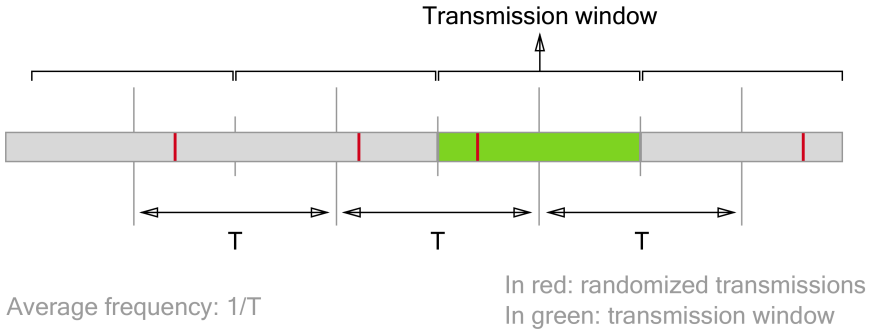

Figure 6.5: Transmission events randomized inside a transmission window

The benefits and drawbacks from this implementation are:

- It does not require consensus between the quadcopters.
- It does not require additional logic to handle the scenario where a quadcopter joins or leaves the swarm.
- It does not avoid packet collisions, which reduces the maximum effective ranging frequency of the ranging scheme.
- The transmission time of the first quadcopter (t_{txA}) has to be sent in the *STATUS* packet, because the order of the responses is not guaranteed.

This implementation is the chosen one because the benefits it provides are more significant than the drawbacks. In addition, the drawbacks are acceptable[4].

The frequency of the randomized transmissions determines the total ranging frequency of the swarm. It should be the minimum possible in order to avoid collisions, but enough to provide accurate positioning results. An average value of 400 transmitted packets for the whole swarm is configured: each quadcopter regulates its transmission frequency according to the number of detected neighbours. In addition:

- A maximum limit of 100 transmissions per quadcopter is configured, which avoids sending too many packets when the swarm is small (under 4 quadcopters).
- A minimum limit of 20 transmissions per quadcopter is configured, in order to guarantee that the maximum measured round time is smaller than the requirement of 0.0672s[5] when the swarm is big (over 20 quadcopters):

$$t_{roundMax} = \frac{1}{20} = 0.05 \ seconds \tag{6.4}$$

[4]Refer to section 7.4 for more information
[5]Refer to section 6.4.2 for the calculation of this value

6.6 Ranging scheme: implementation

This section explains the implementation details of the swarm ranging scheme.

The swarm ranging scheme introduced in section 5.2.2 requires round and reply times in order to calculate TOF. When thinking about the implementation for the first time, the immediate solution appears to be:

- The transmitting quadcopter stores the timestamp for its last transmission, so it can calculate the round time when receiving *STATUS* packets from the neighbours.
- The neighbours calculate their reply time an send it inside their *STATUS* packets.

The following figure illustrates this implementation:

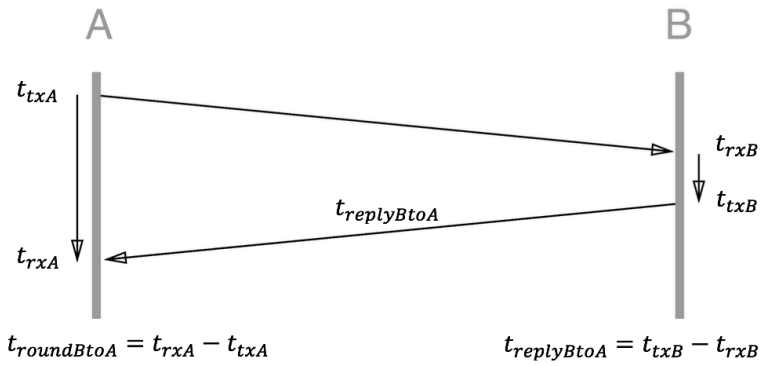

Figure 6.6: First implementation approach for timestamp exchange

Taking a second look at the ranging scheme, this straight away solution is not feasible for the following reasons:

- To obtain the clock correction between quadcopters, the neighbours are required to send an absolute timestamp in order to be able to calculate $t_{sinceLastPacketX}$ ($t_{replyBtoA}$ is not an absolute timestamp, it is the difference between two of them)[6].
- Due to the randomization of the transmission times, t_{txA} can not be kept in memory and must be sent in the *STATUS* packet from A[7].

Thus, the following is implemented:

- When a neighbour B sends its *STATUS* packet, the t_{txB} is the timestamp included in the *STATUS* packet and used to calculate $t_{sinceLastPacketB}$.
- Due to the randomized transmissions, t_{txA} is included in the *STATUS* packet from A, and also included in the *STATUS* packet from B, so A is able to calculate $t_{roundBtoA}$.

[6]Refer to section 6.4.2 for more information
[7]Refer to section 6.5 for more information

- Instead of sending $t_{replyBtoA}$, the neighbour B sends t_{rxB}, so quadcopter A can calculate $t_{replyBtoA}$ using t_{txB} and t_{rxB} (this is an optional modification).

The following figure illustrates this implementation:

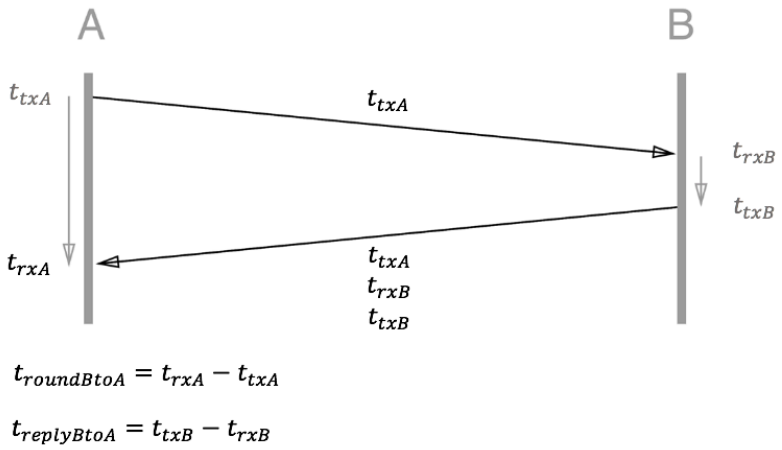

Figure 6.7: Final implementation approach for timestamp exchange

6.7 *STATUS* packet content

This section describes what is the final data traveling in the *STATUS* packet, and what is its size.

Section 3.1 of the DW1000 User Manual [38] describes the content of a packet, and its size depending on the chip configuration. The data to be transmitted has a maximum size of 127 bytes, which is a requirement of the IEEE 802.15.4-2011 UWB standard [40] (Decawave offers a proprietary non standardized option to extend this to 1023 bytes, but it has not been used).

The final declaration of the struct holding the packet data is:

```
typedef struct {
  lpsSwarmPacketHeader_t header;

  payload_t payload[(127 - sizeof(lpsSwarmPacketHeader_t)) /
  ↪  sizeof(payload_t)];
} __attribute__((packed)) lpsSwarmPacket_t;
```

Code Block 6.2: The *STATUS* packet declaration

It contains two main parts: the header and the payload array.

- Header: contains information related with the status of the quadcopter transmitting. The declaration is as follows:

  ```
  typedef struct {
    locoId_t sourceId;
    uint8_t type;

    uint8_t seqNr;
    packetTimestamp_t tx;

    uint8_t payloadLength;
  } __attribute__((packed)) lpsSwarmPacketHeader_t;
  ```

  Code Block 6.3: The header declaration for the *STATUS* packet

  Its elements are:

  – `locoId_t sourceId`: the id of the quadcopter transmitting. It allows to identify it among the rest of the neighbours. Its size is 1 byte (can take values from 0 to

255)[8].

- `uint8_t type`: the type of the packet. It allows to differentiate the packets that are used by the swarm ranging scheme from other packets used by other applications. Its size is 1 byte (can take values from 0 to 255)[9].
- `uint8_t seqNr`: the sequence number assigned to the packet. It is used to discard packets that arrive to the antenna multiple times as a result of reflections. Its size is 1 byte (can take values from 0 to 255)[10].
- `packetTimestamp_t tx`: the transmission time for the packet. Use by the receiving neighbours, to calculate the TOF between the transmitting quadcopter and themselves. Its size is 4 bytes[11].
- `uint8_t payloadLength`: the length of the payload array, used for extracting and processing the payload information. Its value indicates the amount of neighbours' data included in the payload (not the amount of bytes). Its size is 1 byte (can take values from 0 to 255).

The total size of the header is 8 bytes.

- Payload array: contains information about the data that the transmitting quadcopter has about its neighbours. its declaration is as follows:

```
typedef struct {
  locoId_t id;
  packetTimestamp_t rx;
  packetTimestamp_t tx;
  uint16_t tof;
} __attribute__((packed)) payload_t;
```

Code Block 6.4: The payload declaration for the *STATUS* packet

Its elements are:

- `locoId_t id`: the id of the quadcopter that owns the data of the payload. Its size is 1 byte (can take values from 0 to 255)[12].
- `packetTimestamp_t rx`: the reception time for the last *STATUS* packet coming from the quadcopter owning the data of this payload, which uses it to calculate the TOF between the transmitting quadcopter and itself. Its size is 4 bytes[13].

[8] Refer to section 6.2 for more information
[9] Refer to section 6.1 for more information
[10] Refer to chapter 4 for more information
[11] Refer to section 6.4 for information about its size, and section 6.6 for information about how its value is used in the ranging scheme
[12] Refer to section 6.2 for more information
[13] Refer to section 6.4 for information about its size, and section 6.6 for information about how its value is used in the ranging scheme

- `packetTimestamp_t tx`: the transmission time for the last *STATUS* packet coming from the quadcopter owning the data of this payload, which uses it to calculate the TOF between the transmitting quadcopter and itself. Its size is 4 bytes [14].
- `uint16_t tof`: the last calculated TOF from the transmitting quadcopter to the quadcopter owning the data of this payload. Used by the receiving neighbours, to be able to trilaterate the position of the transmitting quadcopter. Its size is 2 bytes, which is enough to represent a distance of up to 307m[15].

  The total size of one payload is 11 bytes.

  Considering the maximum number of bytes that can travel in a packet (127 bytes) and the size of the header (8 bytes), the maximum length of the payload array at compile time is 10. This length sets the maximum number of neighbour data that can be transmitted in each *STATUS* packet.

This structs are packed in order the avoid extra padding bytes being transmitted: the DW1000 driver does not distinguish between padding and data when preparing the `lpsSwarmPacket_t` struct for transmission[16].

The total size of the `lpsSwarmPacket_t` struct is 118 bytes.

[14] Refer to section 6.4 for information about its size, and section 6.6 for information about how its value is used in the ranging scheme

[15] Refer to section 5.3.1 for information about how its value is used in the ranging scheme

[16] Refer to section 6.10 for more information about the packet attribute

6.8 Position estimation: kalman filter

This section explain the implementation of the position calculation logic introduced in section 5.3.

Trilaterating the 3D position of a tag from the position of several neighbours and the distance to them is a conceptually simple task: it requires to find the intersection between multiple spheres. In practise, the implementation hides great complexity due to the following reasons[17]:

- The value of the distances are obtained using sensors that introduce error to the measurement. The following figure illustrates this issue in 2D:

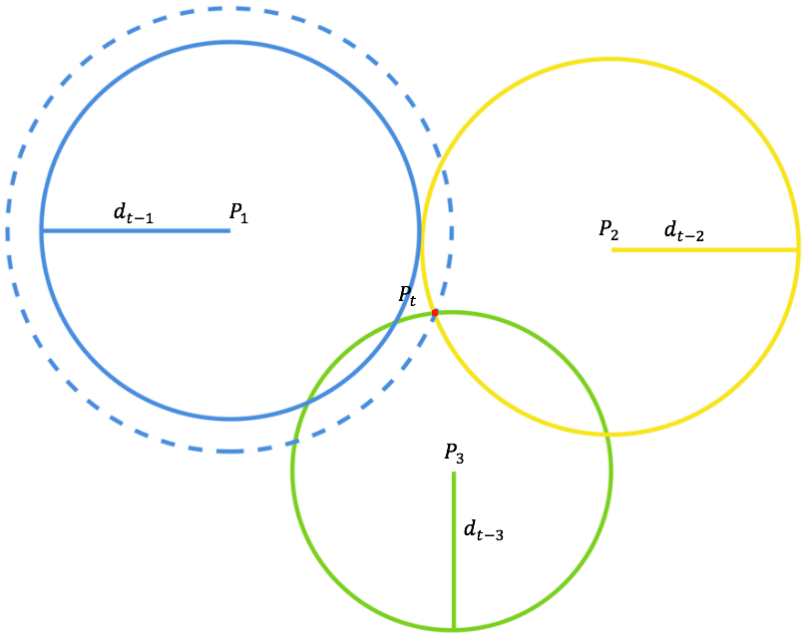

Figure 6.8: Introducing error in the measurement of d_{t-1} makes it impossible to calculate P_t. In dotted lines, the real distances. In full lines the measured distances.

The intersections of the full circumferences do not result in a unique point anymore.

- When ranging, the distance measurements are updated one by one with each *STATUS* packet that is received. This does not work well together with the triangulation equations, which require all the neighbour positions and distances between them to be computed and ready in order to calculate the position of the tag.

In order to solve this issues, the crazyflie firmware already includes the implementation of a kalman filter, which is described in the research paper "Fusing ultra-wideband

[17]Refer to section 2.4 for more information

range measurements with accelerometers and rate gyroscopes for quadrocopter state estimation" [46].

> The kalman filter is used as a tool to trilaterate the position of a quadcopter from the ranging measurements. This chapter talks about its usage and internal functionality. However, its mathematical foundation will not be discussed in detail, as it is not the goal of this master thesis and the knowledge the author has about it is limited. For further information refer to [46].

This implementation allows to:

- Estimate the position of a tag even when the measurement includes error.
- Pass to the filter new measurements at any time, getting an updated position estimation for every *STATUS* packet received.
- Calculate the position estimations using a reduced amount of computer power and memory, such that it is possible to execute it on-board the quadcopter.

In addition, because the position has to be estimated for each one of the neighbours, there has to be one kalman filter per neighbour. This presents the following problems:

- The existent kalman filter is implemented statically, which means that all the variables that keep its internal storage are unique. This limits the number of kalman filters that can be executed at the same time to one. In order to have one filter per neighbour, refactoring is required.
- The number of filters that can be running at the same time will be limited by the computational power and memory available[18].

6.8.1 Kalman filter refactoring

The first step in order to use the kalman filter as a tool to trilaterate the position of the neighbours is refactor its code, so its internal state is not declared as static anymore, allowing multiple instantiations of it. From now on, the old implementation will be called static and the new implementation will be called non-static. The non-static implementation is divided in two:

- The `estimatorKalmanStorage`: the storage includes the variables that keep the internal state of the kalman filter. They are grouped under the `estimatorKalmanStorage_t` struct. Also, in order to reduce complexity, the storage variables for accelerometer and gyroscope related measurements are removed, as they are not used for this application. Its total size is 448 bytes.

[18] Refer to section 7.2 for more information

- The `estimatorKalmanEngine`: all the functions performing operations on the storage are rewritten, so the storage is passed as an argument. As an example, compare the static and non-static implementation of the function returning the estimated position:

```
void estimatorKalmanGetEstimatedPos(point_t* pos) {
  pos->x = S[STATE_X];
  pos->y = S[STATE_Y];
  pos->z = S[STATE_Z];
}
```

Code Block 6.5: The function returning the estimated position, as implemented in the static kalman filter (`estimator_kalman.c` file)

```
static void getPosition(const estimatorKalmanStorage_t* storage,
↪  point_t* position) {
  position->x = storage->S[STATE_X];
  position->y = storage->S[STATE_Y];
  position->z = storage->S[STATE_Z];
}
```

Code Block 6.6: The function returning the estimated position, as implemented in the non-static kalman filter (`estimatorKalmanEngine.c` file)

In addition, all the public API is grouped under the `estimatorKalmanEngine_t` struct. This allows to use it in an object oriented way. Its declaration is as follows:

```
extern const estimatorKalmanEngine_t estimatorKalmanEngine;
```

Code Block 6.7: The non-static kalman filter engine (`estimatorKalmanEngine.h` file)

A detailed comparison of the static and non-static implementations can be found in appendix B.

As a result of this modifications it is possible to instantiate one kalman filter per neigh-

bour[19], and use it as follows:

```
estimatorKalmanEngine.getPosition(&neighboursStorage[i].filter,
↪  &positionData);
```

Code Block 6.8: An example of a call to the non-static kalman filter API

6.8.2 Kalman filter: prediction step

The operations performed by the kalman filter are devided in two steps:

- Prediction step: it uses the velocity (inferred from the changes in the position), angular velocity (measured by the gyroscope) and acceleration (measured by the accelerometer) for predicting the state of the quadcopter at a given time.
  This step also involves a covariance correction substep that increases the values of the covariance matrix: the more time passed since the last update the more inexact is the prediction. It is based on a mathematical model of the quadcopter for better results, and is explained on [47].
- Update step: uses information measured by sensors (such as position, velocity or angular position) to update the state of the quadcopter. The measurements include its standard deviation, which the filter uses to increase or decrease the covariance.

The implementation has shown that using the kalman filter for calculating the relative position between the quadcopters does not solve the position drifting problem described in section 5.3.3, but makes it worse. The main reason behind it is the prediction step:

- In an absolute positioning system, having wrong estimations due to the prediction step is not an important issue, as they are corrected during the update step, when the real measurements are processed.
- In a relative positioning system, having wrong estimations due to the prediction step leads to a wrongly calculated position for that neighbour, which is then used as a reference to calculate the position of other neighbours. As a result, the error introduced by the prediction step of one of the neighbours is extended to the rest of the quadcopters in the swarm.

Concretely, in this implementation what is happening is:

- The position of the quadcopter A is calculated from the position of the neighbours around (assuming there is only one, let's call it B) and the distance measured with UWB signals:

$$P1_A = f(P1_B, d1_{AtoB}) \tag{6.5}$$

[19] Refer to appendix D for the source code of the `neighbourData_t` struct containing the kalman filter storage

- The small errors in the distance introduced by successive UWB measurements ($UWBe_A$) lead to small variations of the position of the quadcopter, even if this quadcopter is stationary. This position variations allow the kalman filter to infer a velocity that has an absolute value higher than zero:

$$V1_A = f(UWBe1_A) \tag{6.6}$$

- Such a velocity is used in the prediction step to calculate a new position:

$$P1_A* = f(P1_A, V1_A) \tag{6.7}$$

- This new position is used as a reference to calculate the positions of the rest of the quadcopters around:

$$P2_B = f(P1_A*, d2_{AtoB}) = f(P1_A, d2_{AtoB}, V1_A) \tag{6.8}$$

- Because of the position change of the neighbours due to the wrongly calculated $P1_A*$, their kalman filters will infer a velocity:

$$V1_B = f(V1_A, UWBe1_B) \tag{6.9}$$

- Such a velocity is used in their prediction steps to calculate a new position:

$$P2_B* = f(P2_B, V1_B) \tag{6.10}$$

- The position of the quadcopter A is calculated, again, from the position of the neighbours around:

$$P2_A = f(P2_B*, d3_{AtoB}) = f(P2_B, d3_{AtoB}, V1_A, UWBe1_B) \tag{6.11}$$

- The change in position allows the kalman filter to infer a new velocity:

$$V2_A = f(V1_B, UWBe2_A) = f(V1_A, UWBe1_B, UWBe2_A) \tag{6.12}$$

As a result, the prediction step makes the error introduced by the UWB measurements stay in the system and induces velocity to the whole swarm.

In order to avoid this as much as possible, the covariance correction step is taken out of the prediction step. Then, the prediction step is disabled, executing only the covariance correction and update steps. As a result, the kalman filter is able to trilaterate the position of the quadcopter while handling the errors introduced by the UWB measurements, and avoiding the position drifting problem introduced in section 5.3.3.

6.8.3 Kalman filter stabilization

As explained before, the kalman filter is used as a tool to estimate the position of the neighbours without solving the complex trilateration equations. This works well when the kalman filter is already running, but not during initialization: it requires an initial value for position and velocity. Which values should be used, if computing the result of the trilateration equations is not an option?

The implemented solution feeds the kalman filter an initial arbitrary value of *(0, 0, 0)* for position and velocity, with the maximum allowed standard deviation. As a result, the covariance matrix stores very high values, indicating that the estimated position and velocity are not stable neither reliable. Then, new measurements (distances to the neighbours) are continuously passed to the kalman filter, rapidly modifying the initial arbitrary values for position and velocity with more realistic estimations. Also, the covariance values become smaller as more measurements are processed.

The values estimated by the kalman filter are used only when they are considered to be stable, which means that the covariance value for the estimation is smaller than a threshold. The chosen threshold has been obtained by experimenting with different values, and is the same as the covariance assigned to the distance measurements: 0.25[20].

6.8.4 Final implementation

This section explains the final implementation for position estimation:

- Initially, the position of a quadcopter is unknown.
- The quadcopter starts discovering other neighbours. Each detected neighbour is associated with a non-static kalman filter, which is used for calculating its position. The data passed to the filter is different depending of the following two scenarios:
  1. If the coordinate system is being built, the data passed to the kalman filter is a combination of real distance measurements to the other neighbours, plus arbitrary values[21].
  2. If the coordinate system is already built, the data passed to the kalman filter is the measured distances to the other neighbours.
- The quadcopter's position is calculated using the static kalman filter. Using the static instead of the non-static implementation is required, because the code is coupled with other systems (for example, it receives data from the accelerometer and gyroscope). The data passed to the filter is a combination of the position and distance to the neighbours.

[20]Refer to section 7.3 for more information
[21]Refer to section 5.3.2 for more information

Every time a new packet is received, an update of the positions of the neighbours and the quadcopter is performed as follows:

1. Every received packet has the identifier of the transmitting neighbour. The TOF values coming in the packet are associated with the positions of the rest of the neighbours, and passed to the non-static kalman filter associated with the transmitting neighbour. As a result, and after the kalman filter stabilizes, the transmitting neighbour gets a fixed position which is calculated relative to the position of the other discovered neighbours (or arbitrary assigned, in case the coordinate system is under construction).
2. If the position of the transmitting neighbour is updated, it is used to update the position of the quadcopter: it is passed to the static kalman filter, together with the calculated distance.

The following figure illustrates the process:

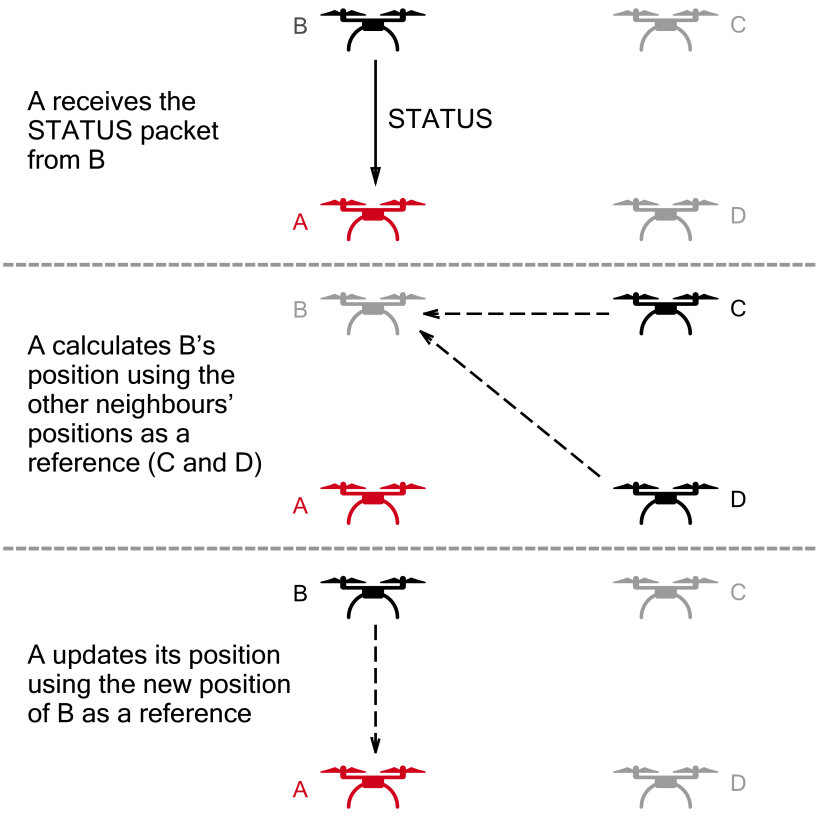

Figure 6.9: Process for updating the position of the quadcopters

6.9 Memory usage

This section describes the data structures used for storing data, its size and the way the memory is managed.

6.9.1 Used data structures

The variables used to store the information required by the swarm ranging scheme are grouped in three main structs:

- The neighbours storage: used to keep information about the neighbours around. Each quadcopter maintains one storage, which contains multiple `neighbourData_t` structs: one per neighbour. This structs store the following information:
  - The tx timestamp coming inside the the last packet received from the neighbour, used for calculating the clock correction between the quadcopter and the neighbour.
  - The rx timestamp of the last packet received from the neighbour, used for calculating the clock correction between the quadcopter and the neighbour.
  - The required data to filter the calculated clock correction between the quadcopter and the neighbour[22].
  - The required data to keep track of the sequence number of the packets received from the neighbour[23].
  - The necessary data for the position trilateration: this is the storage of the kalman filter[24].
- The TOF storage: used to store the calculated TOF between neighbours. Each quadcopter maintains one storage, which contains multiple `tofData_t` structs: one per pair of neighbours. The total required size of this storage depends on the number of quadcopters in the swarm, and can be calculated as follows:

$$size_{tofStorage} = \frac{size_{neighbourStorage} * (size_{neighbourStorage} - 1)}{2} \tag{6.13}$$

- The context struct: each quadcopter keeps its own unique context struct, which is used to maintain its internal state. This struct stores the following information:
  - The neighbour storage.
  - The TOF storage.
  - Its own quadcopter identifier[25].
  - The required data to keep track of the sequence number used when transmitting[26].

[22]Refer to section 6.4 for more information
[23]Refer to section 6.3 for more information
[24]Refer to section 6.8.1 for more information
[25]Refer to section 6.2 for more information
[26]Refer to section 6.3 for more information

– The required data for calculating when to transmit[27].

The final implementation of this structs can be found in appendix D. Their sizes are:

- `neighbourData_t` struct: 488 bytes, with the main part (448 bytes) used by the kalman filter storage.
- `tofData_t` struct: 6 bytes.
- `ctx` struct: 8552 bytes.

6.9.2 Implementation using dynamic allocation: libdict

As explained in section 4.2, there are two ways in which memory can be allocated: static and dynamic. The swarm ranging scheme, because it handles an arbitrary number of neighbours, has memory requirements that change at run time, depending on the size of the swarm. For this reason, the initial implementation focused on using dynamic memory allocation for the neighbour and TOF storages, with the goal of using only the required memory.

Dynamic allocation of memory is something supported by most of the languages used for server, web or mobile development. They include the necessary data structures to work with dynamic allocation of memory: arrays of dynamic length, dictionaries, sets... In the case of the crazyflie, the development language is C: it allows to allocate dynamic memory, but does not offer such data structures. Thus, some third party libraries exist that alleviate this. For this specific implementation, because the ranging data needs to be associated with each quadcopter in the swarm, a dictionary is the best fit.

When looking for a third party library, it is important to consider if it fits certain requirements. A comprehensive list can be found in [48], and some of the main ideas are:

- The library must be maintained: it should be used by a big enough group of people and receive updates periodically.
- It should be well documented and provide sample code.
- It should be reasonably covered by tests and present a low amount of reported bugs.

After this considerations, the chosen library is libdict [49]. It provides:

- Nine different data structures to choose from, which offer different performances in terms of insert, find, and delete routines.
- A common dictionary-like interface to all of them.
- Possibility to use statically or dynamically allocated keys and values.

[27] Refer to section 6.5 for more information

- Demo and benchmarking code.

In order to use it in the crazyflie it is required that:

- The code successfully cross-compiles for the crazyflie.
- The implementation returns valid results for common dictionary operations.
- The speed and memory usage are acceptable.

In order to verify if this requirements are met, it is necessary to benchmark the library while running in the quadcopter.
The benchmarking code included in the repository does not cross-compile for the crazyflie because it includes platform-specific code used for input/output (configure the tests and show the results) or time measurements (used for calculating the benchmark results). Custom tests need to be written.

6.9.2.1 Libdict: test implementation

The custom implemented tests and benchmarks are as follows:

1. A test that dynamic allocates key and value and inserts them in the dictionary. The key is an `uint8_t` type, and the value is a `char*` (an string). After that, it searches for the key, and if found, extracts the value. then, compares the extracted value and the original inserted one. if both are the same, it will return successfully.

```
bool dict_test_dynamic_uint8_insert_search(dictionary_t type)
```

2. A benchmark that dynamic allocates keys of type `unsigned int` and inserts them in the dictionary. It does that until the heap is filled, and returns the total number of executed iterations. This gives an idea of the amount of elements that can be stored in a dictionary before filling the memory. The value of the inserted key is increased by one every iteration to avoid key duplication, and because of that, an `unsigned int` type is needed in order to avoid overflow.

```
int dict_benchmark_dynamic_uint_keys_insert_memory(dictionary_t
↪  type)
```

3. A benchmark that dynamic allocates keys of type `unsigned int`, inserts them in the dictionary and then clears the dictionary. It does that during a specific number of milliseconds, and returns the total number of executed iterations. It is necessary to clear the dictionary to avoid filling the limited memory of the crazyflie before the benchmark finishes. The value of the inserted key is increased by one every iteration to avoid key duplication, and because of that, an `unsigned int` type is needed in order to avoid overflow.

```
int dict_benchmark_dynamic_uint_keys_insert_speed(dictionary_t
↪  type, int ms);
```

4. A test similar to the number 1, but allocating key and value statically.

```
bool dict_test_static_uint8_insert_search(dictionary_t type)
```

5. A test similar to the number 4, but performing an extra step: it uses the key pointer from the last step and increases its value (obtaining a different key value), which is then used to perform another search on the dictionary. The result of this search should return nothing (the key passed this time to the dictionary has a value that none of the keys already inside the dictionary have), in order to consider it a successful test.

```
bool dict_test_static_uint8_insert_search_changing_key_value(
↪  dictionary_t
↪  type)
```

6. A test similar to the number 3, but allocating the keys statically.

```
int dict_benchmark_static_uint_keys_insert_speed(dictionary_t type,
↪  int ms);
```

6.9.2.2 Libdict: test results

In order to obtain reliable results of the benchmarks and tests, they are executed five times, and the results are averaged when needed. The raw data can be seen in appendix C. After analyzing the results, the hash table with open addressing is the data structure that performs better and is the chosen one. Some other highlights are:

1. Test `dict_test_dynamic_uint8_insert_search`: the test is passing for all data structures, meaning that insertion and search work correctly when using keys and values dynamically allocated.
2. Benchmark `dict_benchmark_dynamic_uint_keys_insert_memory`: the results show that the memory usage performs similar for every data structure. The hash table with open addressing obtains the best results.
3. Benchmark `dict_benchmark_dynamic_uint_keys_insert_speed`: here is where the main differences arise between data structures, due to their different implementations. The hash table with open addressing obtains the best results, which shows that it is 46% faster than the next fastest data structure (path reduction tree), and 141% faster than the slowest one (skip list).

4. Test `dict_test_static_uint8_insert_search`: the test is passing for all data structures, meaning that insertion and search work correctly when using keys and values statically allocated.
5. Test `dict_test_static_uint8_insert_search_changing_key_value`: this test throws surprising results. All the data structures tested are failing, except for the two hash table implementations. After some time looking into it, it was not possible to understand the reason. The main hypothesis is that the underlying implementation is different in the way it is treating the key used for insertion and search. No more time was invested troubleshooting this issue, as the hash table with open addressing (the chosen data structure) is passing the test successfully.
6. Benchmark `dict_benchmark_static_uint_keys_insert_speed`: similarly to the `dict_benchmark_dynamic_uint_keys_insert_speed` test, the hash table with open addressing obtains the best results, which shows that it is 171% faster than the next fastest data structure (red black tree), and 502% faster than the slowest one (skip list).

6.9.3 Implementation using static allocation

The development continued using dynamic allocation for several months, which was performing properly. After, the implementation was changed to use static arrays. The main reasons behind this change were:

- All the code in the firmware relies on static allocation, which implies that none of the processes will require to use more memory at run time than the one reserved at compile time. The free RAM left at compile time will continue to be free at run time: using static allocation and reserving more memory than what is required is not a problem, as that memory will not be used anyway.
- Using dynamic allocation means that a heap overflow can happen if the number of neighbours increases enough. Keeping track of the free heap and avoid the overflow is possible (by using specific functions from FreeRTOS) but will require extra logic. Switching to static allocation is easier and safer, as will allow to know at compile time the maximum number of neighbours according to the available memory.

The API for accessing elements from both the neighbour and TOF storages is:

```
tofData_t* findTofData(tofData_t storage[], const locoId_t id1, const
↪  locoId_t id2, const bool insertIfNotFound);
unsigned int countTof(tofData_t storage[]);
neighbourData_t* findNeighbourData(neighbourData_t storage[], const
↪  locoId_t id, const bool insertIfNotFound);
unsigned int countNeighbours(neighbourData_t storage[]);
void removeOutdatedData(neighbourData_t neighbourDataStorage[],
↪  tofData_t tofDataStorage[]);
```

Code Block 6.9: API for accessing elements from both the neighbour and TOF storage

The function `removeOutdatedData` is the one in charge or managing quadcopters not responding or leaving the swarm. It uses the `IsValid` variable of the `neighbourData_t` struct[28] to identify any neighbours that did not send any *STATUS* packets during one second. When this happens, their storage and associated TOF values are removed.

[28]Refer to appendix D for the source code

6.10 Memory alignment

This section describes the root cause of an issue found during development: accessing the value of a packed struct from a new pointer was randomly causing the application to throw an exception, and the quadcopter to reboot.

6.10.1 Unaligned access

The STM includes a Cortex-M4 core, which has 32 bits width registers. This means that when it is reading or writing from memory, it makes so in groups of 4 bytes.
For example, consider reading an integer of 32 bits from memory. Two scenarios can happen:

0x0	0x1	0x2	0x3
0x4	0x5	0x6	0x7
0x8	0x9	0xA	0xB

Figure 6.10: Aligned 32 bits integer

0x0	0x1	0x2	0x3
0x4	0x5	0x6	0x7
0x8	0x9	0xA	0xB

Figure 6.11: Unaligned 32 bits integer

- In figure 6.10, the integer first address is multiple of 4 bytes (address `0x4`): this means that is 4-bytes aligned. Because it is aligned, the MCU can read the whole integer in one attempt.
- In figure 6.11, the integer first address is not multiple of 4 bytes (address `0x5`): this means that it is misaligned. Because it is not aligned, the MCU has to read first the addresses `0x4`, `0x5`, `0x6` and `0x7`, then the addresses `0x8`, `0x9`, `0xA` and `0xB` and then move the content of addresses `0x5`, `0x6` and `0x7`, `0x8` to another register in order to be able to operate with the integer.

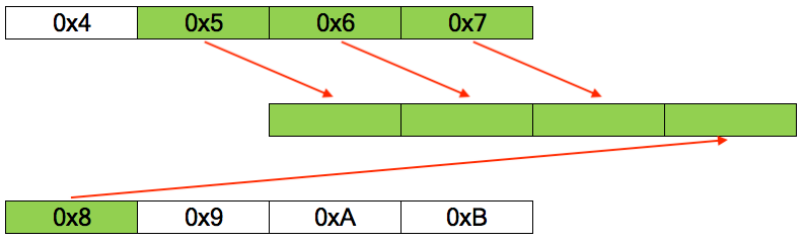

Figure 6.12: Reading an unaligned 32 bits integer

In older processors, this operations have to be synthesized in software by doing a series of small accesses, and combining the results. This introduces an important speed penalty. Since the ARMv6 architecture [50] introduced in 2002, there is also hardware support for unaligned access. This improves the speed, but is still slower than aligned access [51].

In addition, some languages allow the developer control over the alignment. The C

language introduces several type attributes that can be used with struct and union types for this purpose. One of them is the packed attribute.

6.10.2 The packed attribute

The packed attribute is introduced in the following lines due to its relevance during the development for embedded systems, and its application in the crazyflie project. "This attribute, attached to struct or union type definition, specifies that each member (other than zero-width bit-fields) of the structure or union is placed to minimize the memory required" [52]. This disables the alignment for the members inside this types. For example, consider the following struct, with and without the packed attribute:

```
struct test {
  char a;
  int b;
  char c;
};
```

Code Block 6.10: Normal struct

```
struct test {
  char a;
  int b;
  char c;
} __attribute__((packed));
```

Code Block 6.11: Packed struct

The corresponding memory layout would be:

Figure 6.13: Normal struct memory layout

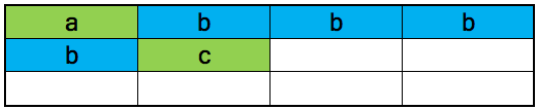

Figure 6.14: Packed struct memory layout

- Because the integer requires 4 bytes alignment, when allocating memory for a normal struct the compiler has to add padding bytes after `char a`.
- When using the packed attribute, the compiler does not align the integer: all the elements of the struct are placed one after the other in memory, without padding.

Using the packed attribute has the following benefits and drawbacks:

- The memory is used more efficiently because the members are located one after another, without introducing padding.
- The memory access is slower because the MCU has to perform the operations shown in figure 6.12.
- Access to the members of a packed struct or union must be done directly from a variable of the containing type. Creating a new pointer to a member must never be done because the compiler assumes that the pointer is aligned: dereferencing it will result

in an aligned access to unaligned data, leading to a run time exception [53]. When using the clang compiler it is possible to pass the `-Waddress-of-packed-member` flag in order to generate warnings when this happens. Sadly, in the GCC compiler this flag does not exist.

CHAPTER 7
Results

This chapter shows the results and performance of the system after the implementation details discussed in chapter 6.

7.1 Position estimation accuracy

This section shows the accuracy of the position estimation. The following results are obtained from a swarm of three quadcopters performing relative positioning in 2D. The figure below illustrates the scenario:

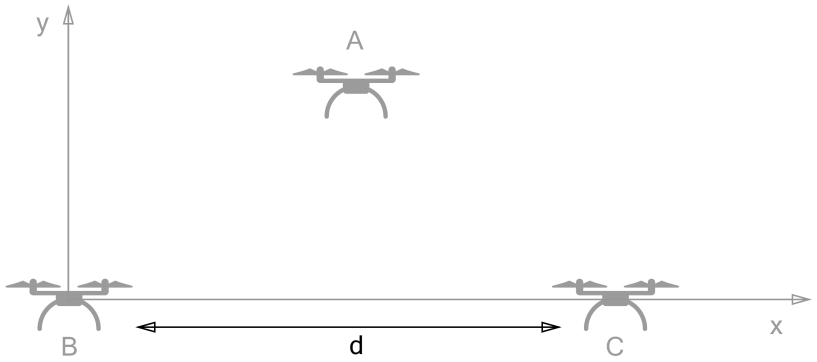

Figure 7.1: The scenario used to obtain the results. The XY plane in this figure represents an horizontal surface (a table), and all the quadcopters on it are placed on the surface (Z position = 0)

The data is obtained from quadcopter A, which is moved to four locations:

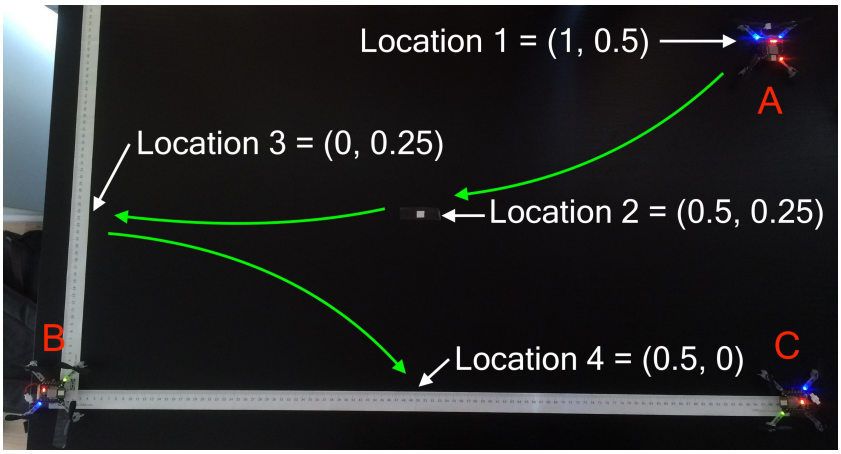

Figure 7.2: The four locations of the recorded data

In every location 10 seconds of data are recorded, with a frequency of 100 samples per second, resulting in 1000 samples per location. The procedure to obtain the data is as follows:

1. The quadcopters B and C are placed at (0, 0) and (1, 0) respectively. Quadcopter A is placed at location 1.
2. Quadcopter A is turned on first, quadcopter B is turned on second, and quadcopter C is turned on third. This order ensures that when creating the coordinate system, B is given the (0, 0) position, and C is given the (d, 0) position.
3. Data is obtained for location 1.
4. Quadcopter A is moved to location 2. Data is obtained for location 2.
5. Quadcopter A is moved to location 3. Data is obtained for location 3.
6. Quadcopter A is moved to location 4. Data is obtained for location 4.

The comparison between the real positions and the average of the estimated positions for each location is shown in the tables below:

Quad	X real	X estimated	Y real	Y estimated	X Std Dev	Y Std Dev
A	1	0.828784067	0.5	0.600036183	0.06276467	0.1086374
B	0	-0.011811921	0	0	0.0054372	0
C	1	0.853015914	0	0	0.003362	0

Table 7.1: Real VS estimated position values for location 1

Quad	X real	X estimated	Y real	Y estimated	X Std Dev	Y Std Dev
A	0.5	0.433466549	0.25	-0.003676294	0.01774367	0.01538889
B	0	-0.002391461	0	0	0.00342396	0
C	1	0.888804187	0	0	0.00366274	0

Table 7.2: Real VS estimated position values for location 2

Quad	X real	X estimated	Y real	Y estimated	X Std Dev	Y Std Dev
A	0	-0.295414913	0.25	0.047723681	0.10787378	0.08283415
B	0	0.0170965	0	0	0.01041691	0
C	1	0.917776924	0	0	0.00247512	0

Table 7.3: Real VS estimated position values for location 3

Quad	X real	X estimated	Y real	Y estimated	X Std Dev	Y Std Dev
A	0.5	0.450998304	0	0.353764279	0.05313401	0.07583487
B	0	0.03595113	0	0	0.00557871	0
C	1	0.936423364	0	0	0.00713261	0

Table 7.4: Real VS estimated position values for location 4

7.2 CPU load and memory usage

This section shows performance measurements of CPU load and memory usage.

7.2.1 CPU load

The following figure shows the CPU load:

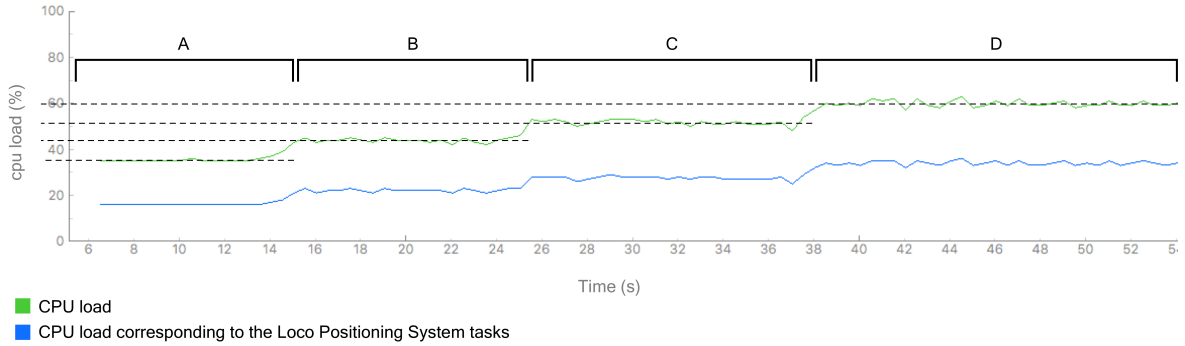

Figure 7.3: CPU load for a quadcopter during normal operation, while running the swarm ranging scheme

The two lines of the figure show the total CPU load, and how much of that load is due to the ranging and position operations of the swarm ranging scheme (the LPS is the one in charge of that). Because the kalman filters are the main load for the CPU, it is possible to observe that the variation of load coming from the LPS has a great impact in the overall CPU load.

In order to analyze the figure, it has been divided in the following parts:

- Section A: shows the load when there are zero neighbours around. The 35% CPU load is the result of the transmission attempts of the ranging scheme, plus the operation of the rest of the systems (FreeRTOS, radio communication stack...).
- Section B: shows the load when there is one neighbours around. The 9% increase in CPU load is due to the processing of incoming packets, the operation of the non-static kalman filter that calculates the position of the neighbour and the operation of the static kalman filter that calculates the position of the quadcopter.
- Section C: shows the load when there are two neighbours around. The 7% increase in CPU load is due to the processing of a higher number of incoming packets and the operation of the non-static kalman filter for the new neighbour. In total, there are three kalman filters executing at the same time: two non-static for calculating the position of each neighbour and one static for calculating the position of the quadcopter.
- Section D: shows the load when there are three neighbours around. The 9% increase in CPU load is due to the processing of a higher number of incoming packets and

the operation of the non-static kalman filter for the new neighbour. In total, there are four kalman filters executing at the same time: three non-static for calculating the position of each neighbour and one static for calculating the position of the quadcopter.

It is possible to observe that the increase of CPU load when a new neighbour is added is not constant, but grows after the first quadcopter. This is because:

- Every time a new neighbour is added to the swarm, a new kalman filter is initiated. This represent a constant increase of CPU load.
- For every new quadcopter more data is fed to the existing kalman filters. This results in a CPU load increase that is higher the more neighbours are in the swarm.

7.2.2 Memory usage

As explained in section 6.9, all the implementation of the ranging scheme uses statically allocated memory. This allows to know the memory usage at compile time. Passing the `--print-memory-usage` flag to the linker at build time prints the different memory regions and their usage.

Building a version of the firmware without the swarm ranging scheme implementation shows the following memory usage:

```
Memory region         Used Size  Region Size  %age Used
             RAM:       78152 B       128 KB     59.63%
           FLASH:      168776 B      1008 KB     16.35%
```

Code Block 7.1: Memory usage without the swarm ranging scheme

For the actual implementation, which configures a maximum of 16 tracked neighbours (with their associated 16 kalman filters) the memory usage is as follows:

```
Memory region         Used Size  Region Size  %age Used
             RAM:       88560 B       128 KB     67.57%
           FLASH:      178568 B      1008 KB     17.30%
```

Code Block 7.2: Memory usage when configuring a maximum of 16 tracked neighbours

The maximum number of neighbours that can be tracked without filling the RAM is 72.

As can be seen from this results, the main part of the RAM usage is due to the other

systems implemented on the firmware: only approximately 8% of it is due to the swarm ranging scheme[1]. From it:

- 82% corresponds to the `ctx` struct, which contains the neighbour and TOF storages.
- 75% corresponds to the neighbour storage, which contains 16 `neighbourData_t` structs.
- 69% corresponds to the 16 non-static kalman filters calculating the neighbour's position (16 `estimatorKalmanStorage_t` structs).
- 7% corresponds to the TOF storage, which contains 120 `tofData_t` structs.

[1]Refer to sections 6.9.1 and 6.8.1 for the exact size of the structs

7.3 Standard deviation adjustment

This section explains how to calculate the value assigned to the distance measurements' standard deviation.

As explained in 6.8, the kalman filter is in charge of managing the error introduced by the TOF measurements. In order to do that, every measure passed to the filter has an associated standard deviation. The higher the standard deviation is, the less reliable the measure is considered.

The following figures show the relative positioning of two quadcopters in a one dimension space, where the position of each of them is calculated from the position of the other. The first one shows the results when the standard deviation is 0:

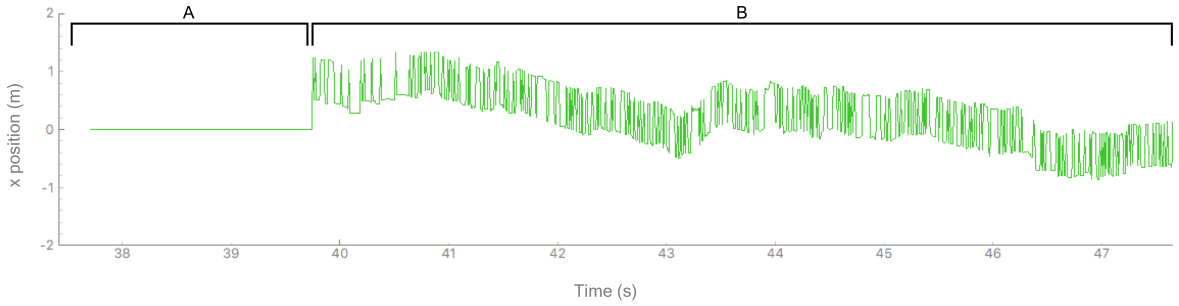

Figure 7.4: 1D relative position calculation for two quadcopters. The standard deviation value is 0

In order to analyze the figure, it has been divided in the following parts:

- Section A: the first neighbour joins the swarm, and its position is set to 0.
- Section B: the second neighbour joins the swarm. After this moment, the position of both neighbours are calculated relative to each other.

It is possible to observe how the position of both quadcopters is not stable and moves up and down, while the distance between them is approximately the same all the time. This is the result of position drifting problem[2].

[2]Refer to section 5.3.3 for more information

The second figure shows the positioning when the standard deviation is 0.25:

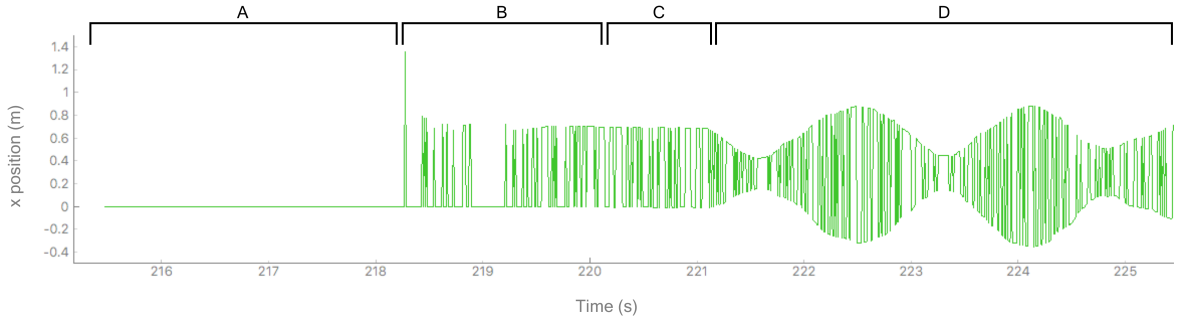

Figure 7.5: 1D relative position calculation for two quadcopters. The standard deviation value is 0.25

In order to analyze the figure, it has been divided in the following parts:

- Section A: the first neighbour joins the swarm, and its position is set to 0.
- Section B: the second neighbour joins the swarm. During this time, the position of the first neighbour is 0, and multiple measurements are passed to the kalman filter of the second neighbour. This happens until it stabilizes[3].
- Sections C and D: the kalman filter of the second neighbour is stable. From now on, the position of both neighbours are calculated relative to each other.

Section C shows the position of both neighbours when the distance between them is not changing. Section D shows their position when one of the quadcopters is being moved closer and further from the other. Observe that the position of both quadcopters is now constant, and does not move up or down, solving the position drifting problem introduced in section 5.3.3.

Observe that in section D, one of the quadcopters is not moving, but still the figure shows changes on the position of of both of them. This is expected behaviour: because the position calculation is done from measurements of the distances between the neighbours, when that distance changes it is not possible to know which of them was the one moving. As a result, the position is adjusted on both neighbours equally.

[3]Refer to section 6.8.3 for more information

7.4 Ranging frequency: performance

This section shows experimental results for the performance of the ranging scheme implementation described in section 6.5.

The following figure highlights information about the packet exchange when using the swarm ranging scheme:

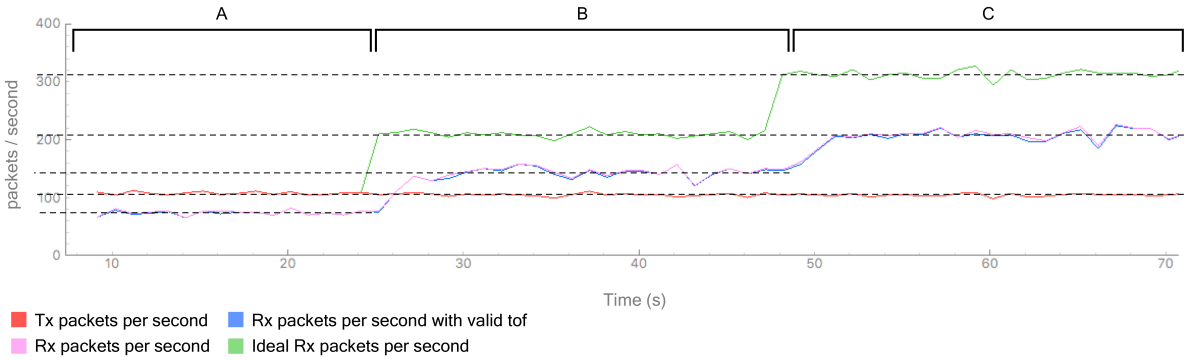

Figure 7.6: Number of sent and received packets during normal operation

In order to analyze the figure, it has been divided in the following parts:

- Section A: the swarm has two neighbours.
- Section B: a new neighbour joins the swarm: the swarm has three neighbours.
- Section C: a new neighbour joins the swarm: the swarm has four neighbours.

The green line shows the "ideal Rx packets per second", which is the result of multiplying the actual tx packets per second (red line) by the number of neighbours in the swarm. Comparing the green line with the pink line it is possible to obtain the following performance results:

- Section A:
$$performance = 100 * \frac{75}{105} = 71 \tag{7.1}$$
- Section B:
$$performance = 100 * \frac{140}{210} = 66 \tag{7.2}$$
- Section C:
$$performance = 100 * \frac{210}{315} = 66 \tag{7.3}$$

Performance results show that approximately 1/3 of the packets are lost during transmission. In addition, comparing the blue line and the pink line it is possible to observe that almost 100% of the packets received have a valid TOF value: it is considered valid if it represents a distance between 0 and 10 meters, which is useful for rejecting faulty measurements.

CHAPTER 8
Conclusions

This chapter analyses the results shown in chapter 7.

- In terms of the position estimation accuracy, the results from section 7.1 show that:
  - The smaller are the distances between the quadcopters, the worse is the estimated position.
  - The estimations on the Y axis are worse than on the X axis. The two reference neighbours (B and C) used for calculating the relative position define the X axis. The lack of other neighbours with positions far from the X axis is suspect to be the the main cause for this lack of precision.
  - The position of quadcopters B and C is stable and does not drift[1].

  The main reason for the inaccurate results is suspected to be introduced by the antenna delay correction[2].
- The results from section 7.2 show that the CPU load establish a limitation on the maximum number of neighbours that can be tracked and used for positioning. From this data it is safe to say that 3D positioning is possible (a minimum of 4 neighbours are required), but tracking a higher number of quadcopters in order to get better precision will rapidly lead to 100% CPU usage.
- The results from section 7.3 show that using kalman filters without the prediction step to filter the distance measurements provides good results: allows fine adjustment of the amount of filtering by modifying the value of the standard deviation.
- The results from section 7.4 show that randomizing the transmissions is a good enough alternative for ranging: it greatly simplifies the logic needed for managing the scenario when quadcopters join/leave the swarm, while the lost of packets is acceptable.

The results obtained from a 2D implementation show that a decentralized solution that allows each quadcopter in a swarm to know about the distances between the neighbours is feasible in terms of CPU, memory and required ranging frequency. Initial tests show that the distance measurements include errors from multiple sources (antenna delay, packet loss, packet delay...) that when used for position trilateration greatly reduce the accuracy of the system[3].

The implementation could not be tested in a 3D scenario due to the lack of time. The results of such tests will show if a higher number of quadcopters can improve the precision of the system, and if 3D positioning is possible and accurate enough for using it with swarm of quadcopters.

[1] Refer to section 5.3.3 for more information
[2] Refer to section 6.4 for more information
[3] As seen in section 7.1

The source code developed during this master thesis can be found in [54].

CHAPTER 9
Future work

As explained in chapter 8, 3D tests of the positioning system could not be conducted due to lack of time, but they have the highest priority in terms of future work. After them, the areas of this master thesis that would benefit from further investigation are the following:

- The antenna delay is corrected by using an arbitrary value, which is the same for every quadcopter. This is suspected to be the main source of error in the UWB measurements. It is worth investigating what are the possible alternatives for adjusting such value without requiring individual calibration of each chip.
- The computational power required to operate the kalman filter is limiting the maximum number of quadcopters per swarm. It may be possible to reduce it by avoiding repetitive floating point calculations, and instead store and reuse their result. This will require slightly more memory, but that is not a limitation at the moment.
- While ranging, approximately 30% of the packets are being lost. This number could probably be reduced by adjusting the transmission window and maximizing the time that the DW1000 is listening for incoming packets. It is also possible to implement more elaborated randomization schemes, such as the ones resulting from the research around the ALOHA protocol.

Bibliography

[1] Chun Fui Liew et al. "Recent Developments in Aerial Robotics: A Survey and Prototypes Overview". In: *CoRR* abs/1711.10085 (2017). eprint: 1711.10085. URL: http://arxiv.org/abs/1711.10085.

[2] Intel. *Intel Falcon 8+ System*. [Accessed April of 2018]. 2017. URL: https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2017/05/intel-falcon-8-plus-system-brochure.pdf.

[3] DJI. *MAVIC Pro User Manual*. [Accessed April of 2018]. 2017. URL: https://dl.djicdn.com/downloads/mavic/Mavic%20Pro%20User%20Manual%20V2.0-.pdf.

[4] Vicon. [Accessed April of 2018]. URL: https://www.vicon.com.

[5] OptiTrack. [Accessed April of 2018]. URL: https://optitrack.com.

[6] The Straits Times Lester Hio. *Loss of GPS signal grounded NDP drones*. [Accessed August of 2018]. URL: https://www.straitstimes.com/singapore/loss-of-gps-signal-grounded-ndp-drones.

[7] Nick Iliev and Igor Paprotny. "Review and comparison of spatial localization methods for low-power wireless sensor networks". In: *IEEE Sensors Journal* 15.10 (2015), pages 5971–5987.

[8] Jiang Xiao et al. "A Survey on Wireless Indoor Localization from the Device Perspective". eng. In: *Acm Computing Surveys* 49.2 (2016), page 25. ISSN: 15577341, 03600300. DOI: 10.1145/2933232.

[9] Fangzheng Zhou. "A Survey of Mainstream Indoor Positioning Systems". In: *Journal of Physics: Conference Series*. Volume 910. 1. IOP Publishing. 2017, page 012069.

[10] Wilson Sakpere, Michael Adeyeye-Oshin, and Nhlanhla B.W. Mlitwa. "A state-of-the-art survey of indoor positioning and navigation systems and technologies". eng. In: *South African Computer Journal* 29.3 (2017), pages 145–197. ISSN: 23137835, 10157999. DOI: 10.18489/sacj.v29i3.452.

[11] Abdulrahman Alarifi et al. "Ultra wideband indoor positioning technologies: Analysis and recent advances". In: *Sensors* 16.5 (2016), page 707.

[12] Fazeelat Mazhar, Muhammad Gufran Khan, and Benny Sällberg. "Precise Indoor Positioning Using UWB: A Review of Methods, Algorithms and Implementations". In: *Wireless Personal Communications* 97.3 (2017), pages 4467–4491.

[13] K. A. Horváth, G. Ill, and Á. Milánkovich. "Passive extended double-sided two-way ranging algorithm for UWB positioning". In: *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*. July 2017, pages 482–487. DOI: 10.1109/ICUFN.2017.7993831.

[14] A. I. Baba. "Calibrating Time of Flight in Two Way Ranging". In: *2011 Seventh International Conference on Mobile Ad-hoc and Sensor Networks*. December 2011, pages 393–397. DOI: 10.1109/MSN.2011.23.

[15] Jianga Shang et al. “Improvement schemes for indoor mobile location estimation: A survey”. In: *Mathematical Problems in Engineering* 2015 (2015).

[16] André G Ferreira et al. “Performance Analysis of ToA-Based Positioning Algorithms for Static and Dynamic Targets with Low Ranging Measurements”. In: *Sensors* 17.8 (2017), page 1915.

[17] Bitcraze. *Crazyflie 2.0 decks*. [Accessed August of 2018]. URL: https://www.bitcraze.io/2015/03/crazyflie-2-0-decks/.

[18] Bitcraze. *Crazyflie 2.0 hardware specifications*. [Accessed August of 2018]. URL: https://store.bitcraze.io/products/crazyflie-2-0.

[19] Bitcraze. *Crazyflie 2.0 System Architecture*. [Accessed August of 2018]. URL: https://wiki.bitcraze.io/projects:crazyflie2:architecture:index.

[20] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, December 2011, 683 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

[21] Free Software Foundation Inc. *Linker Scripts*. [Accessed August of 2018]. URL: https://sourceware.org/binutils/docs/ld/Scripts.html.

[22] Nordic Semiconductor. *S110 nRF51822. SoftDevice Specification v1.3*. [Accessed August of 2018]. URL: http://infocenter.nordicsemi.com/pdf/S110_SDS_v1.3.pdf.

[23] Bitcraze. *Crazyflie 2.0 nRF51 Master Boot Switch repository*. [Accessed August of 2018]. URL: https://github.com/bitcraze/crazyflie2-nrf-mbs.

[24] Bitcraze. *Crazyflie 2.0 nRF51 Bootloader repository*. [Accessed August of 2018]. URL: https://github.com/bitcraze/crazyflie2-nrf-bootloader.

[25] Bitcraze. *Crazyflie 2.0 nRF51 Firmware repository*. [Accessed August of 2018]. URL: https://github.com/bitcraze/crazyflie2-nrf-firmware.

[26] Bitcraze. *Crazyflie 2.0 STM32F4 Bootloader repository*. [Accessed August of 2018]. URL: https://github.com/bitcraze/crazyflie2-stm-bootloader.

[27] Bitcraze. *Crazyflie 2.0 STM32F4 Firmware repository*. [Accessed August of 2018]. URL: https://github.com/bitcraze/crazyflie-firmware.

[28] FreeRTOS. *FreeRTOS main webpage*. [Accessed June of 2018]. URL: https://www.freertos.org.

[29] Bitcraze. *Crazyflie LOG subsystem*. [Accessed June of 2018]. URL: https://wiki.bitcraze.io/projects:crazyflie:firmware:log.

[30] ARM. *GNU Arm Embedded Toolchain*. [Accessed June of 2018]. URL: https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads.

[31] Free Software Foundation Inc. *GNU Make*. [Accessed June of 2018]. URL: https://www.gnu.org/software/make/.

[32] Free Software Foundation Inc. *GCC, the GNU Compiler Collection*. [Accessed June of 2018]. URL: https://gcc.gnu.org.

[33] Throw The Switch. *CMock*. [Accessed June of 2018]. URL: https://github.com/ThrowTheSwitch/CMock.

[34] Throw The Switch. *Unity*. [Accessed June of 2018]. URL: https://github.com/ThrowTheSwitch/Unity.

[35] Andrés Cecilia Luque. *Setup a C/C++ makefile project in Xcode*. [Accessed June of 2018]. URL: https://medium.com/@acecilia/setup-a-c-c-makefile-project-in-xcode-syntax-highlight-autocompletion-jump-to-definition-and-80405bd4542e.

[36] RTOS. *Static Vs Dynamic Memory Allocation*. [Accessed May of 2018]. URL: https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html.

[37] Amanjot Kaur Randhawa and Alka Bamotra. *Study of static and dynamic memory allocation*. [Accessed May of 2018]. 2017. URL: http://ijicse.in/wp-content/uploads/2017/06/30.pdf.

[38] DecaWave. *DW1000 user manual*. [Accessed June of 2018]. URL: https://www.decawave.com/sites/default/files/dw1000_user_manual_2.12.pdf.

[39] DecaWave. *DW1000 datasheet*. [Accessed June of 2018]. URL: https://www.decawave.com/sites/default/files/resources/dw1000-datasheet-v2.09.pdf.

[40] "IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)". In: *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)* (September 2011), pages 1–314. DOI: 10.1109/IEEESTD.2011.6012487. URL: https://ieeexplore.ieee.org/document/6012487/.

[41] DecaWave. *DW1000 module datasheet*. [Accessed June of 2018]. URL: https://www.decawave.com/sites/default/files/dwm1000-datasheet-v1.6.pdf.

[42] DecaWave. *APS014 application notes: antenna delay calibration of DW1000 products and system*. [Accessed June of 2018]. URL: https://www.decawave.com/sites/default/files/aps014-antennadelaycalibrationofdw1000-basedproductsandsystems_v1.01.pdf.

[43] Andrés Cecilia Luque. *Added crystal calibration from OTP*. [Accessed August of 2018]. URL: https://github.com/bitcraze/libdw1000/pull/11.

[44] Bitcraze. *Driver for DW1000*. [Accessed August of 2018]. URL: https://github.com/bitcraze/libdw1000.

[45] Norman Abramson. "THE ALOHA SYSTEM: Another Alternative for Computer Communications". In: *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference*. AFIPS '70 (Fall). Houston, Texas: ACM, 1970, pages 281–285. DOI: 10.1145/1478462.1478502. URL: http://doi.acm.org/10.1145/1478462.1478502.

[46] M. W. Mueller, M. Hamer, and R. D'Andrea. "Fusing ultra-wideband range measurements with accelerometers and rate gyroscopes for quadrocopter state estimation". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. May 2015, pages 1730–1736. DOI: 10.1109/ICRA.2015.7139421.

[47] Mark W Mueller, Markus Hehn, and Raffaello D'Andrea. "Covariance Correction Step for Kalman Filtering with an Attitude". In: *Journal of Guidance, Control, and Dynamics* (2016), pages 1–7.

[48] Roadfire Software Josh Brown. *Save your future self from broken apps by asking these questions BEFORE adding a third-party framework to your app*. [Accessed August of 2018]. URL: https://roadfiresoftware.com/2015/08/save-your-future-self-from-broken-apps/.

[49] Farooq Mela. *C library of key-value data structures*. [Accessed August of 2018]. URL: https://github.com/fmela/libdict.

[50] ARM David Brash. *The ARM Architecture Version 6*. [Accessed August of 2018]. URL: https://www.simplemachines.it/doc/ARMv6_Architecture.pdf.

[51] ARM. *How does the ARM Compiler support unaligned accesses?* [Accessed August of 2018]. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka15414.html.

[52] Free Software Foundation Inc. *Specifying Attributes of Types*. [Accessed August of 2018]. URL: https://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html.

[53] ARM. *Packed type attribute*. [Accessed August of 2018]. URL: http://www.keil.com/support/man/docs/armclang_ref/armclang_ref_chr1393328521340.htm.

[54] Andrés Cecilia Luque. *Master Thesis repository*. [Accessed September of 2018]. URL: https://github.com/acecilia/crazyflie-xcode-project.

APPENDIX A
Maximum throughput

This appendix shows the performance and total rangings per second of a UWB communication link in a real world test. The tests make use of two quadcopters sending empty packets between them, without delays, and transmitting back as soon as a packet is received:

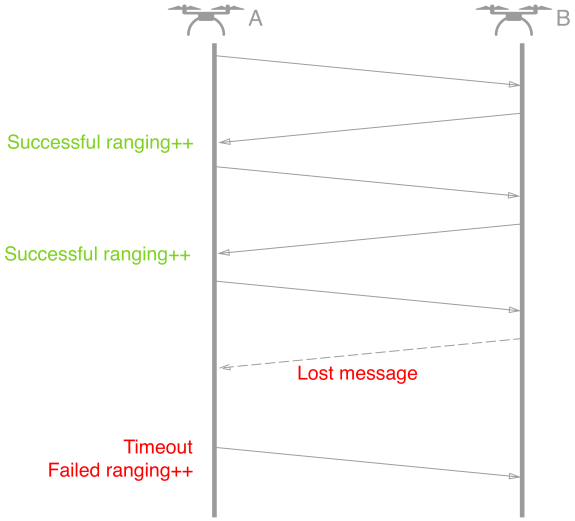

Figure A.1: How the measurement of a successful or failed ranging attempt is implemented

The following results show the performance and total ranging attempts:

$$\text{Performance} = \frac{\text{Successful rangings}}{\text{Total rangings}} \tag{A.1}$$

$$\text{Total rangings} = \text{Successful rangings} + \text{Failed rangings} \tag{A.2}$$

Timestamp (ms)	**Performance**	**Total rangings**
113366	100	1531
114366	100	1535
115366	100	1533
116366	100	1533
117366	100	1533
118366	100	1532
119366	100	1535
120366	100	1533
121366	100	1534
122366	99	1531
123366	100	1532
124366	100	1532
125366	99	1534
126366	100	1531
127366	100	1533
128366	100	1534
129366	100	1533
130366	100	1533
131366	99	1533
132366	100	1531
133366	99	1531
134366	100	1534
135366	99	1531
Average	**99.7826087**	**1532.695652**

Table A.1: Real measurement of the maximum throughput per second

APPENDIX B

Static VS non-static implementation of the kalman filter

B.1 Storage

- Static implementation:

```
// The quad's state, stored as a column vector
typedef enum
{
  STATE_X, STATE_Y, STATE_Z, STATE_PX, STATE_PY, STATE_PZ, STATE_D0, STATE_D1,
   ↪  STATE_D2, STATE_DIM
} stateIdx_t;

static float S[STATE_DIM];

// The quad's attitude as a quaternion (w,x,y,z)
// We store as a quaternion to allow easy normalization (in comparison to a rotation
 ↪  matrix),
// while also being robust against singularities (in comparison to euler angles)
static float q[4] = {1,0,0,0};

// The quad's attitude as a rotation matrix (used by the prediction, updated by the
 ↪  finalization)
static float R[3][3] = {{1,0,0},{0,1,0},{0,0,1}};

// The covariance matrix
static float P[STATE_DIM][STATE_DIM];
static arm_matrix_instance_f32 Pm = {STATE_DIM, STATE_DIM, (float *)P};


/**
 * Internal variables. Note that static declaration results in default initialization
 ↪  (to 0)
 */

static bool isInit = false;
static bool resetEstimation = true;
static int32_t lastPrediction;
static int32_t lastBaroUpdate;
static int32_t lastPNUpdate;
static Axis3f accAccumulator;
static float thrustAccumulator;
static Axis3f gyroAccumulator;
static baro_t baroAccumulator;
static uint32_t accAccumulatorCount;
static uint32_t thrustAccumulatorCount;
static uint32_t gyroAccumulatorCount;
static uint32_t baroAccumulatorCount;
static bool quadIsFlying = false;
static int32_t lastTDOAUpdate;
static float stateSkew;
static float varSkew;
static uint32_t lastFlightCmd;
```

```
static uint32_t takeoffTime;
static uint32_t tdoaCount;
```

Code Block B.1: Original code from the static kalman filter (`estimator_kalman.c` file) showing the variables that store the internal state

- Non-static implementation:

```
// The quad's state, stored as a column vector
typedef enum
{
  STATE_X, STATE_Y, STATE_Z, STATE_PX, STATE_PY, STATE_PZ, STATE_D0, STATE_D1,
   ↪  STATE_D2, STATE_DIM
} stateIdx_t;

/*
 * The struct keeping the internal storage for the kalman estimator. Declare it
 ↪  statically: using it in any other way may cause stack overflow or fill the
 ↪  available dynamic memory
 */
typedef struct {
  float S[STATE_DIM];

  // The queues to add data to the filter
  xQueueHandle accelerationDataQueue;
  xQueueHandle angularVelocityDataQueue;
  xQueueHandle positionDataQueue;
  xQueueHandle distanceDataQueue;
  xQueueHandle velocityDataQueue;

  // The quad's attitude as a quaternion (w,x,y,z)
  // We store as a quaternion to allow easy normalization (in comparison to a rotation
   ↪  matrix),
  // while also being robust against singularities (in comparison to euler angles)
  float q[4];

  // The quad's attitude as a rotation matrix (used by the prediction, updated by the
   ↪  finalization)
  float R[3][3];

  // The covariance matrix
  float P[STATE_DIM][STATE_DIM];
  arm_matrix_instance_f32 Pm;

  // Internal variables
  bool isInit;
  uint32_t lastUpdate;
} estimatorKalmanStorage_t;
```

Code Block B.2: Original code from the non-static kalman filter (`estimatorKalmanStorage.h` file) showing the variables that store the internal state

B.2 API

- Static implementation:

```
void estimatorKalmanInit(void);
bool estimatorKalmanTest(void);
void estimatorKalman(state_t *state, sensorData_t *sensors, control_t *control, const
 ↪  uint32_t tick);

/**
 * The filter supports the incorporation of additional sensors into the state estimate
 ↪  via the following functions:
 */
bool estimatorKalmanEnqueueTDOA(tdoaMeasurement_t *uwb);
bool estimatorKalmanEnqueuePosition(positionMeasurement_t *pos);
bool estimatorKalmanEnqueueDistance(distanceMeasurement_t *dist);
bool estimatorKalmanEnqueueTOF(tofMeasurement_t *tof);
bool estimatorKalmanEnqueueAsoluteHeight(heightMeasurement_t *height);
bool estimatorKalmanEnqueueFlow(flowMeasurement_t *flow);

/*
 * Methods used in the optical flow implementation to get elevation and reset position
 */
float estimatorKalmanGetElevation();
void estimatorKalmanSetShift(float deltax, float deltay);

void estimatorKalmanGetEstimatedPos(point_t* pos);
```

Code Block B.3: Original code from the static kalman filter (`estimator_kalman.h` file) showing its API

- Non-static implementation:

```
typedef struct {
  /**
   Constants defined and used inside the estimator implementation. Externalized here in
 ↪  case they are needed
   */
  float maximumAbsolutePosition;        // In meters
  float maximumAbsoluteVelocity;        // In m/s

  void (*init)(estimatorKalmanStorage_t* storage, const vec3Measurement_t*
   ↪  initialPosition, const vec3Measurement_t* initialVelocity, const
   ↪  vec3Measurement_t* initialAttitudeError);
  void (*update)(estimatorKalmanStorage_t* storage, bool performPrediction);

  // Incorporation of additional data
  bool (*enqueueAcceleration)(const estimatorKalmanStorage_t* storage, const Axis3f*
   ↪  acceleration);
  bool (*enqueueAngularVelocity)(const estimatorKalmanStorage_t* storage, const Axis3f*
   ↪  angularVelocity);
  bool (*enqueuePosition)(const estimatorKalmanStorage_t* storage, const
   ↪  positionMeasurement_t* position);
  bool (*enqueueDistance)(const estimatorKalmanStorage_t* storage, const
   ↪  distanceMeasurement_t* distance);
```

```
  bool (*enqueueVelocity)(const estimatorKalmanStorage_t* storage, const measurement_t*
   ↪  velocity);

  bool (*isPositionStable)(const estimatorKalmanStorage_t* storage, const float
   ↪  maxStdDev);
  bool (*isVelocityStable)(const estimatorKalmanStorage_t* storage, const float
   ↪  maxStdDev);

  void (*getPosition)(const estimatorKalmanStorage_t* storage, point_t* position);
  void (*getState)(const estimatorKalmanStorage_t* storage, state_t* state);
} estimatorKalmanEngine_t;

extern const estimatorKalmanEngine_t estimatorKalmanEngine;
```

Code Block B.4: Original code from the non-static kalman filter (`estimatorKalmanEngine.h` file) showing its API

APPENDIX C

Libdict test results

The test show the performance of the libdict library executing in the quadcopter. Each of them is executed five times. The results are as follows:

1. Test `dict_test_dynamic_uint8_insert_search`:

Data structure	Passed (Ok / Failed)
height_balanced	Ok
path_reduction	Ok
red_black	Ok
treap	Ok
splay	Ok
skip_list	Ok
weight_balanced	Ok
ht_separate_chaining	Ok
ht_open_addressing	Ok

Table C.1: Dynamic insert/search test

2. Benchmark `dict_benchmark_dynamic_uint_keys_insert_memory`:

Data structure	Run 1	Run 2	Run 3	Run 4	Run 5	Average
height_balanced	201	201	201	201	202	201.2
path_reduction	201	201	201	201	202	201.2
red_black	201	201	201	201	202	201.2
treap	201	201	201	201	202	201.2
splay	201	201	201	201	202	201.2
skip_list	198	198	198	198	199	198.2
weight_balanced	201	201	201	201	202	201.2
ht_separate_chaining	173	173	173	173	173	173
ht_open_addressing	209	209	209	209	212	209.6

Table C.2: Dynamic insert memory benchmark (higher is better)

3. Benchmark `dict_benchmark_dynamic_uint_keys_insert_speed`:

Data structure	Run 1	Run 2	Run 3	Run 4	Run 5	Average
height_balanced	170071	157393	171311	171249	157571	165519
path_reduction	146412	172196	157738	157859	166432	160127.4
red_black	159620	170436	172326	163801	160362	165309
treap	151615	139077	161630	149597	143152	149014.2
splay	150311	162299	145641	157125	161962	155467.6
skip_list	97056	96160	97044	100019	94998	97055.4
weight_balanced	165336	147838	161277	172151	147651	158850.6
ht_separate_chaining	135005	146640	137053	141952	148326	141795.2
ht_open_addressing	236508	227053	237540	246078	226167	234669.2

Table C.3: Dynamic insert speed benchmark (executed during 2000ms, higher is better)

4. Test `dict_test_static_uint8_insert_search`:

Data structure	Passed (Ok / Failed)
height_balanced	Ok
path_reduction	Ok
red_black	Ok
treap	Ok
splay	Ok
skip_list	Ok
weight_balanced	Ok
ht_separate_chaining	Ok
ht_open_addressing	Ok

Table C.4: Static insert/search test

5. Test `dict_test_static_uint8_insert_search_changing_key_value`:

Data structure	Passed (Ok / Failed)
height_balanced	Failed
path_reduction	Failed
red_black	Failed
treap	Failed
splay	Failed
skip_list	Failed
weight_balanced	Failed
ht_separate_chaining	Ok
ht_open_addressing	Ok

Table C.5: Static insert/search test, reusing same key pointer but changing its value

6. Benchmark `dict_benchmark_static_uint_keys_insert_speed`:

Data structure	Run 1	Run 2	Run 3	Run 4	Run 5	Average
height_balanced	272220	256000	257659	257819	265021	261743.8
path_reduction	263213	277581	256280	256546	290297	268783.4
red_black	291015	270096	286462	286244	286350	284033.4
treap	262224	241212	261710	263233	218131	249302
splay	245548	289977	237689	262214	276457	262377
skip_list	127635	129050	125616	129145	129068	128102.8
weight_balanced	272799	250036	272703	273914	265066	266903.6
ht_separate_chaining	198823	231865	198845	214473	237798	216360.8
ht_open_addressing	777375	764436	777174	776479	764058	771904.4

Table C.6: Static insert speed benchmark (executed during 2000ms, higher is better)

APPENDIX D

Storage: neighbour, TOF and context

This appendix shows the structs used for storing the data used for the swarm ranging scheme.

- Neighbours storage: the `neighbourData_t` struct.

```
typedef struct {
  bool isInitialized;
  locoId_t id;
  bool IsValid;

  packetTimestamp_t localRx;
  packetTimestamp_t remoteTx;

  clockCorrectionStorage_t clockCorrectionStorage;
  uint8_t expectedSeqNr;

  estimatorKalmanStorage_t estimator;
} neighbourData_t;
```

Code Block D.1: The declaration of the struct used for storing neighbours data

- TOF storage: the `tofData_t` struct.

```
typedef struct {
  bool isInitialized;
  locoIdx2_t id;

  uint16_t tof;
} tofData_t;
```

Code Block D.2: The declaration of the struct used for storing TOF data

- Quadcopter storage: the `ctx` struct.

```
static struct {
  neighbourData_t neighboursStorage[NEIGHBOUR_STORAGE_CAPACITY];
  tofData_t tofStorage[TOF_STORAGE_CAPACITY];

  locoId_t localId;
  uint8_t nextTxSeqNr;

  uint32_t timeOfNextTx;
  uint32_t averageTxDelay;

  bool isBuildingCoordinateSystem;

  xTimerHandle timer;
} ctx;
```

Code Block D.3: The declaration of the struct used for storing the internal state of quadcopter

{2.7182818284}ϑωερτυθιοπσδφγηξκλ