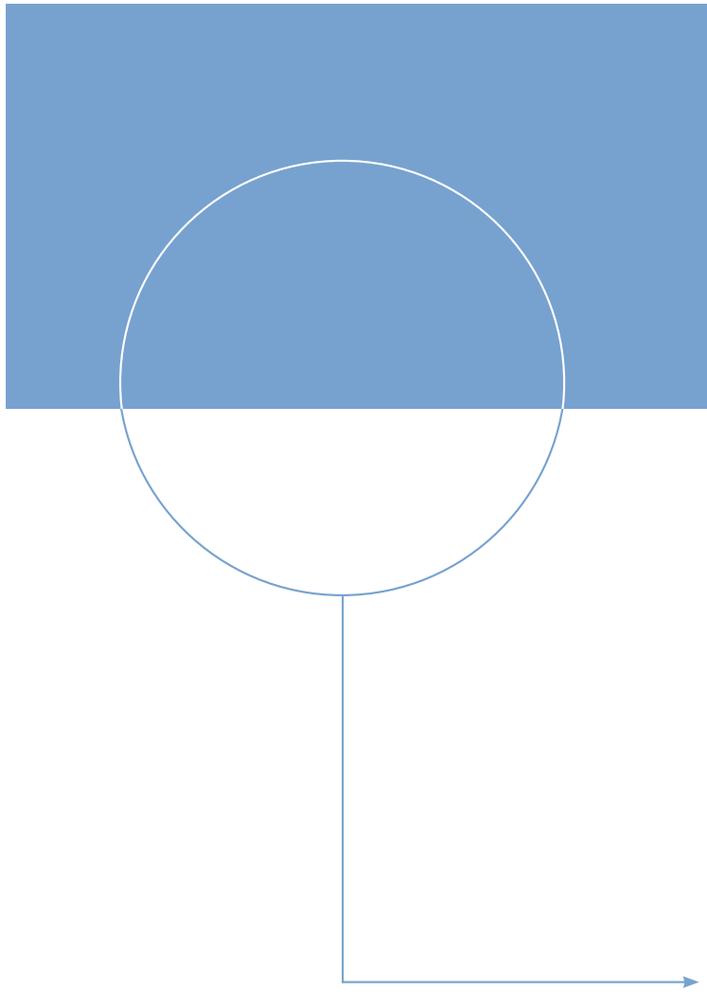


Master's thesis

2019

Per Arne Kjelsvik

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Engineering Cybernetics

Per Arne Kjelsvik

Internet of Fish

Real-time monitoring of fish through LPWAN and Internet technologies

July 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University of
Science and Technology

Internet of Fish

Real-time monitoring of fish through LPWAN and Internet technologies

Per Arne Kjelsvik

Cybernetics and Robotics
Submission date: July 2019
Supervisor: Jo Arve Alfredsen
Co-supervisor: Waseem Hassan

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Summary

Aquaculture is an ever-increasing important industry worldwide. As scale of operations increases, so does the biomass, the risk, and the cost. The industry faces many challenges today, including problems such as sea lice, algae, site overexposure, and production related diseases. If the industry wants to grow and thrive, a greater understanding of the inner workings of the fish is needed.

Internet of Fish (IoF) is an option to aid in this challenge. IoF is a concept that enables transport of acoustic tag data from subsurface environments to a front-end application, subsequently enabling real-time monitoring. The network architecture of this concept can be defined in three layers: perception, network, and application. Earlier work has developed the perception and network layer of the architecture. The main goal of the thesis was therefore to design and implement application layer software, specifically back-end and front-end code that enabled real-time monitoring of IoF data. Additionally, the study aimed to implement positioning based on time of arrival difference (TDOA), and to design and improve network transport protocol from the perception layer to the application layer.

These goals are considered achieved as the concept was proven and tested with a case study. The project work has shown that the Internet of Fish concept is a viable concept for real-time monitoring of live fish in commercial sea cage situations. A new IoF message protocol was implemented, greatly saving on the amount of data being transmitted. The data collected in the case study has been explored, but not in great depth, as it was only meant to illustrate the power of IoF in the context of this thesis. The culmination of the work done has resulted in a useful tool to have for both commercial implementations and research.

Sammendrag

Akvakultur er en verdensomspennende og voksende industri, og særlig viktig er industrien i Norge. Ved økende omfang blir det samtidig økning i biomasse, risiko og kostnader. Det er mange utfordringer knyttet til til akvakulturindustrien, inkludert sjølus, alger, overeksponering av lokasjoner og produksjonsrelaterte sykdommer. Dersom industrien videre skal blomstre må fisken forstås på et enda grundigere nivå.

Internet of Fish (IoF) er et alternativ som bistå i denne utfordringen. IoF er et konsept som muliggjør transport av akustisk tag data fra undervannsmiljø til en front-end applikasjon, noe som muliggjør sanntidsovervåkning. Nettverksarkitekturen til dette konseptet kan defines i tre lag: persepsjon, nettverk, og applikasjon. Tidligere arbeid har utviklet persepsjon- og nettverk-laget til arkitekturen. Hovedmålet med denne oppgaven var derfor å designe og implementere applikasjonslag-programvare, mer spesifikt back-end og front-end kode som muliggjøre sanntidsovervåkning av IoF data. I tillegg, så var det et mål å implementere posisjonering basert på difference av ankomsttid (TDOA), og å designe og forbedre nettverkstransport-protokoll fra persepsjonslaget til applikasjonslaget.

Disse målene regnes som oppnådd ettersom konseptet var bevist og testet med et case study. Prosjektarbeidet har vist at IoF-konseptet er et passende verktøy for sanntidsovervåkning av levende fisk i kommersielle oppdrettsmerd-situasjoner. En ny IoF meldingsprotokoll ble implementert, svært besvarende i mengde data som blir overført. Dataen samlet i case studyet har blitt utforsket, men ikke i detalj, da det kun var ment til å illustrere mulighetene til IoF i kontekst av denne oppgaven. Kulminasjonen av arbeidet som har blitt gjort har ført til et nyttig verktøy både kommersielt og forskningsrelatert.

Preface

This master's thesis is the culmination of my last year as a master student on the 5-year Cybernetics and Robotics Master of Technology programme at the Norwegian University of Science and Technology (NTNU), first laying the groundwork in my specialization project, and then finishing the complete IoF system Spring semester 2019. Jo Arve Alfredsen was my main supervisor, and Waseem Hassan was my co-supervisor.

NTNU provided me with the tools needed for the software development of this thesis, and my supervisors provided me with equipment to develop and test the system. The development was carried out with the following equipment:

- A Dell Optiplex 7040 computer with 32 GB of RAM, accompanied with a desk and two large screens.
- All software development was written in `python 3.7`.
- Jo Arve provided me with a SLIM module, two tags, and a TBR I could use to develop and test new SLIM software in office.
- Waseem provided me with code developed for the SLIM module, cables to aid in debugging, as well as a gateway for testing locally in the office. He was also of great help, providing me with insight into how the SLIM software was structured.
- The mechanical workshop of ITK provided us with waterproof casings and railings to use for instrumentation of SLIM modules for the case study done during the thesis period.
- Åsmund Stavdahl helped me by setting up a virtual server with a MQTT server running on it. He also made it easy to modify and install my own code, giving access to the server through SSH.
- Paho Eclipse was used for the back-end MQTT client.
- Dash by Plotly was used for the front-end web application implemented during the project.

All components of the IoF system and their respective usage in the network architecture are described in detail in chapters 3 and 4.

The goal of the final system was mostly up to me to decide. I was given a pre-existing system, a SLIM module working with a Thelma BioTel TBRs and tags, and an overall goal: “Make it possible to plot data interactively in a web browser”. The process allowed me to learn how the full IoF system works from the lowest network layer (tags) to the highest (application code). Researching how to best utilize the data from the tags in a high-level application design, choosing which libraries to use, and how to solve the different challenges was entirely up to me.

I would like to thank Jo Arve Alfredsen and Waseem Hassan for their supervision of this project. It has been challenging to make this system, and I have been able to develop individual software writing skills in a great way. I would also like to thank Midt-Norsk Havbruk AS for allowing us to run an experiment in their very exciting Aquatraz cage.

I have always had a fascination for visual mediums and data science, and combining these in order to understand more about the behavior of fish has been a truly interesting task. I am grateful to have been able to take on this assignment. To explore data from real fish in an application I have designed and implemented on my own has been a gratifying experience.

Table of Contents

List of Tables

List of Figures

Abbreviations

CSS	=	Cascading Style Sheets
DST	=	Data Storage Tag
GPS	=	Global Positioning System
HDOP	=	Horizontal Dilution of Precision
IoF	=	Internet of Fish
IoT	=	Internet of Things
JSON	=	JavaScipt Object Notation
LoRa	=	Long Range (LoRaWAN)
LPWAN	=	Low-Power Wide-Area-Network
LSB	=	Least Significant Bits
MQTT	=	Message Queuing Telemetry Transport
MSB	=	Most Significant Bits
NTNU	=	Norwegian University of Science and Technology
PCB	=	Printed Circuit Board
PEP	=	Python Enhancement Proposal
PFF	=	Precision Fish Farming
PLF	=	Precision Livestock Farming
SNR	=	Signal-to-Noise Ratio
SQL	=	Structured Query Language
SSM	=	Surface Support Module
TBR	=	Thelma BioTel hydrophone receiver (TBR 700RT)
TDOA	=	Time Difference of Arrival
UTC	=	Universal Time Coordinated
WAN	=	Wide Area Network

Chapter 1

Introduction

1.1 Problem description

Fish is an important protein source, and with rising health awareness, population increase, overfishing, and advancements in technology and logistics, importance of farmed fish as a protein source will only grow. However, unlike land-based farming, aquaculture is a comparatively young endeavour, and there are several additional challenges related to farming fish both on land and in open sea.

How to **collect continuous individual data from fish with acoustic telemetry, and how this can be enhanced with Internet of Things technology** is a problem that will be explored in this project thesis.

1.2 Problem context

The first conceptual framework of Precision Livestock Farming (PLF) was established in 2004 [5]. This framework is concerned with principles of using automated sensing and detection of farm animal responses, and mathematical modelling of animal behavioural and physiological dynamics. In 2017, a framework for commercial aquaculture inspired by PLF, called Precision Fish Farming (PFF), was introduced [12]. In this framework, operational processes are considered to consist of four phases: Observe, Interpret, Decide

and Act. Figure 1.1 shows a cyclic representation of these phases. Quickly summarized, the observe phase wants to observe animal variables that describe bio-responses. From these animal variables, the following phases want to extract feature variables that can be used as target variables to be acted upon, manipulating system and eliciting desired bio-response. From here the cycle repeats, and subsequently you have the potential for closed-loop feedback applications.

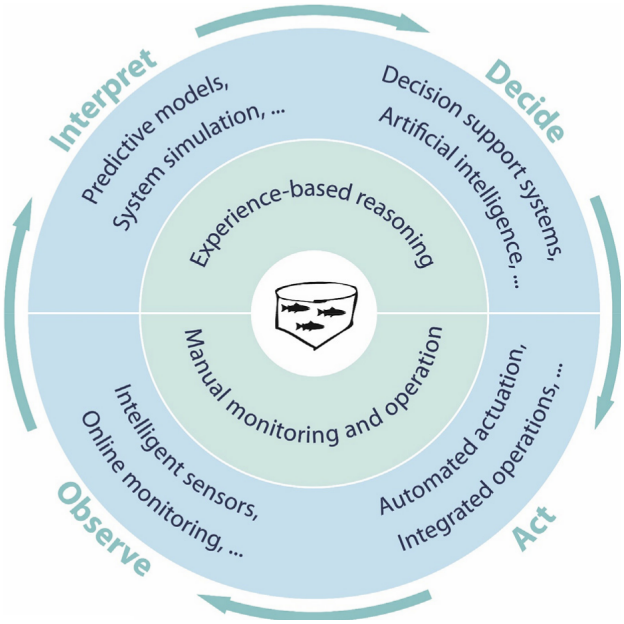


Figure 1.1: A cyclical representation of PFF where operational processes are considered to consist of four phases: Observe, Interpret, Decide and Act. The inner cycle represents the present state-of-the-art in industry, with manual actions and monitoring, and experience-based interpretation and decision-making. The outer cycle illustrates how the introduction of PFF may influence the different phases of the cycle. Figure credits: Andreas Myskja Lien, SINTEF Ocean. Reproduced here with permission.

In the context of PFF, acoustic telemetry is presented as the only viable method for obtaining continuous data series from individual fish in commercial sea-cages, with the added benefit of potentially monitoring physiology such as heart rate and blood composition [12]. The first acoustic telemetry tag equipment for biomonitoring fish was developed in 1956 by the U.S. Bureau of Commercial Fisheries and the Minneapolis-Honeywell Regulator Corporation [39]. Since then several advancements have been made. Today,

acoustic telemetry is a technology that can be used in commercial salmon aquaculture for biomonitoring, and it is favorable to Data Storage Tags (DSTs), as it is able to provide data in real time which enables farmers to respond to the received information with farm management measures [11]. Acoustic telemetry have for example been used to monitor effects of crowding and delousing procedures on Atlantic Salmon (*salmon salar*) [13]. These papers were based on usage in commercial sea cages, but acoustic telemetry for fish monitoring has traditionally been used extensively in wild fish research, where [18], [27], [41] are a few recent examples. With this in mind, it is interesting to explore how this method of observation can be elevated.

Traditionally, the data obtained through acoustic telemetry is logged to a stationary hydrophone in water, and then at a later date, the data is retrieved. From here, researchers can do offline analysis of the time period where acoustic tags were active. For acoustic telemetry to be used as as an observation method in PFF, it is important to facilitate online monitoring of fish. A potential solution to facilitate real-time monitoring is what this article hopes to present, in the form of the IoF concept. This is interesting both to researchers for analysis and monitoring of fish in real-time, as well as commercial sea cage farmers wanting to gain insight into how fish behave on a day-to-day basis.

1.3 Aims of the present study

The Internet of Fish (IoF) alludes to the Internet of Things (IoT) and constitutes a concept where information generated by acoustic sensor transmitters carried by fish, or attached to other underwater objects, is made available for humans and machines through the use of digital hydrophones, LPWAN and Internet technologies. Here we intend to make a first implementation of the IoF concept where novel SLIM units (Synchronization and LoRa Interface Modules) augment the TBR700RT acoustic telemetry receiver with accurate GNSS-based time synchronization and low-power LoRaWAN connectivity, thereby providing a bridge between the wireless underwater sensors and the realm of the Internet. The project includes the following tasks:

- Identify and describe requirements to the IoF system by revisiting the main results

and experiences made in the preceding Specialization Project (Fordypningsprosjekt) [19]. Describe the network architecture and make a refined high-level design of the system from the perception layer to the application layer

- Identify all relevant data types generated by the SLIM/TBR700RT and define corresponding message formats, protocols and network transports from the end-device to the IoF back-end. Select and implement a flexible solution for persistent back-end storage of IoF data.
- Make a working implementation of the full protocol on the SLIM. Data compression methods should be explored and implemented to optimize utilization of LoRaWAN bandwidth.
- Demonstrate IoF functionality by implementing a IoF web-based application front-end for the use-case targeting 2/3D visualization of fish space usage in aquaculture sea-cages.
- Plan and conduct tests to validate the IoF concept with the described use-case.

1.4 Previous work

The IoF concept aims to realize real-time real-time monitoring of fish through a three-layer network architecture: perception, network, and application. An end-device in the perception layer is needed to facilitate communication to the network layer, where a link is needed to forward the received data to the application layer. NTNU PhD candidate Waseem Hassan has developed a perception layer module, the Synchronization and LoRa Interface Module (SLIM), to facilitate the communication link between perception and network. The perception layer and network layer of the concept has been well defined and tested. Recently, two papers he has written on the IoF concept have been accepted for publication, one in-press at time of writing [17], [16]. In the application layer, a software interface is needed to make data easily accessible by end-users. Here, not much work as been done previously. Previous implementations have only stored raw bytes data to `txt`-files, and used `MATLAB` for offline analysis.

A specialization project on the IoF concept was done in the Fall semester 2018 [19]. For the project, back-end software was written to handle the then used SLIM IoF message

format, and a case study was done to validate the system. The case study made it clear that better message compression was needed, and so a design for more efficient compression of IoF data was suggested. Lastly, a simple webpage containing fixed non-interactive plots was created to showcase case study data.

The project was summarized in a report detailing the IoF concept, background theory, network architecture, design and implementation of back-end software, a case study, results from the case study and a short discussion about it. The report concluded with what should be done in future work. Firstly, designing a new IoF protocol based on the suggestions of the report, and implementing said protocol in both SLIM and back-end. Secondly, to include GPS packet as part of the data SLIM sends. Thirdly, to implement TDOA-positioning in back-end for incoming messages. Finally, to implement a fully working front-end web application to access data in real-time. These challenges has been the focus of the development work done during the Master thesis Spring 2019.

This thesis is in other word the continuation of the groundwork done in the specialization project. It was natural to heavily base the overall structure and some of the content of this thesis on the previous report. Several aspects have therefore been kept or only modified, such as the broader context of IoF, the background theory, and the parts of the project that remained the same, namely the network architecture, equipment and tools, and the case study. Most of the thesis time was spent on developing new software, and so the software is the main focus of this thesis. Note that there is no need to read the previous report, as the whole implemented system is fully explained and detailed in this report. This section is only to clarify for the reader what has been done before, and to properly self-reference previous work. The specialization project was never published, and it is not publicly available.

1.5 Structure of the thesis

The article begins with background on principles the IoF concept is built upon in chapter 2. Details about how acoustic transmission work is given in section 2.1. Further, principles of positioning based on acoustic transmission is included in section 2.2, and some definitions of network and communication protocols are given in section 2.3.

In chapter 3 the network architecture requirements of the IoF concept is identified and presented. Structured requirements to the application layer is also defined and presented in ordered lists.

Design and implementation of the IoF concept is presented in chapter 4. The chapter starts with identifying data types and describing the corresponding message formats and protocols for the IoF system. Following that, in section 4.2 a discussion on persistent storage options and solutions is given. Thereafter components of the system are presented in section 4.3 and software tools and libraries utilized in back-end and front-end is listed in section 4.4. Finally, details on how the back-end was implemented, as well as how front-end code functions, is presented in more detail in section 4.5. The IoF source code structure is given in Appendix A, while TDOA positioning and timestamp drift adjustment implementation is given in Appendix B and C.

The project was realized with a case study done in cooperation with Midt Norsk Havbruk AS. The setup and execution of this case study is presented in chapter 5. The case study started during the specialization project period. No other implementation code has been included, but the full source code can be found on a open GitHub repository [20].

The requirements of the IoF application layer defined in chapter 3 is validated in chapter 6, addressing whether each requirements was fulfilled or not. Section 6.3 showcases all the different plot types the implemented front-end web application supported, and section 6.4 presents more figures of the case study data.

Chapter 7 gives a discussion on how well the aims of the study were met, and some discussion is given on the observed data in the case study.

Finally, chapter 8 concludes the findings of the paper, and presents some suggestions to what should be done in future work.

Chapter 2

Background

This chapter presents some background for the principles and research that the work of this thesis is based on. The chapter is divided into three sections:

- Section 2.1 covers components of acoustic fish telemetry, compromises to keep in mind when designing acoustic tags, possibilities of the tags, and differential pulse modulation.
- Section 2.2 presents principles on positioning with hydrophones and acoustic tags, an analytical solution to determine transmission origin, resolving ambiguity of position candidates, and dilution of precision.
- Section 2.3 covers definitions of communication protocols and concepts used in the system, namely Internet of things (IoT), MQTT, and LoRa.

2.1 Acoustic fish telemetry

2.1.1 Components of fish telemetry

First of all, an acoustic tag is needed. As the name implies, this is a small sound-emitting device, designed to send out sound signals in specific frequencies in subsurface environments. The sound transmits omnidirectionally from the tag, and its components

generally consists of a battery, sensor(s), a modulator, and a transmitter. Tags have a couple of desired characteristics that must be kept in mind, as it is important to make some compromises when designing them (section 2.1.2).

Second, to make use of the signals transmitted by a tag, a receiver, or rather a hydrophone, is needed to detect them. The hydrophone will listen for sound signals in a specific frequency or a range of frequencies, converting and storing digitally any recognized signals it perceives within those frequencies. Several types of hydrophones exist with varying use cases. Some simply log the data internally until the hydrophone is retrieved manually. Others can communicate digitally, for example with the RS-485 standard. While historical analysis is useful, real-time data communication is of great value and potential for day-to-day operations and remote monitoring (section 2.1.4).

Third, the data collected needs to be transferred to a device that can be used for monitoring and/or analysis. For a logging hydrophone, the data can be transferred to a desktop computer after retrieval through for example Bluetooth and then analyzed offline. For hydrophones that can communicate, a second option would be to connect it directly to a desktop computer for offline/online viewing, depending on internet access on-site. A third option would be to connect the hydrophone to a device that can forward the data another way, such as low-power radio.

The last two configurations are the only viable solutions for real-time monitoring. Connecting a computer directly to a hydrophone alleviates the need of developing custom hardware. However, in situations where it is desirable to monitor wild fish, neither power, internet or good infrastructure is readily available. The environmental conditions are also generally exposed and challenging at farming sites, making instrumentation difficult: To stretch a cable 200 meters and have it securely connected to a hydrophone is not necessarily desirable or viable. In addition, power outages happen, and internet access can be limited.

Therefore, a more viable option would be to have a wireless device close to the hydrophone, running on battery, forwarding the data through for example low-power radio. It would also need to be designed for low-power usage to optimize battery life time. While the battery would have to be replaced regularly, it is not susceptible to power outages. And it is not limited by internet infrastructure or lack thereof, assuming long distance

low-powered radio is a viable option in the environment it is located. This option of a low-powered wireless device is used for the IoF project.

2.1.2 Compromises to keep in mind when designing tags

Acoustic tags have a couple of desired characteristics, such as low weight, small size, long transmission range, long operating life, low cost, and reliable performance. To meet all of them simultaneously is difficult, so some compromises are to be expected. This in turn often means that individual projects will require customized tags suitable for that project.

Batteries are important in terms of weight and size of the tag, and naturally they determine the length of operation life. The size and weight of the tag determines what species of fish the tags can be used with, and in what stage of life. A small fry should not be instrumented with a large tag, as it will impede the normal functioning of the fish, nullifying the integrity of the data. Smaller batteries will therefore be better suited for smaller fish, but the operation time is consequently shortened.

One way to increase the battery longevity is to use pulsed transmitters rather than continuous transmitters. This way, a fraction of the required powered is needed. Another benefit of transmitting pulses is less disturbance between messages, as the front of each message will be clear (assuming the previous message has had time to break down). How often the tag transmits a message, as well as how much power it uses also impacts the battery life of the tag. Here a trade-off must be made in different projects, whether high update speed is valued for more continuous detection, or if longer-term tracking with less frequent updates is desired.

The material of the tag casing is also important, as it will be inserted inside the fish (tags placed outside the fish also exist). Choosing the wrong materials can potentially lead to reduced welfare and affect fish behaviour. In addition, the insertion of tags requires surgery, which in itself can affect the fish negatively. Welfare is closely linked to behaviour of the fish, and is an important animal variable to keep in mind when designing any type of technology for observation of fish.

2.1.3 Differential pulse modulation

Acoustic tags usually use a differential pulse modulation scheme for improved battery efficiency and transmission. This means that data is contained in the time between pulses, rather than continuous transmission where the pulses are the data.

One way to implement a scheme like this can be seen in fig. 2.1. In this case, the length between the first two pulses determine code type. Several code types can be used, like S256 and R64K, which are standardized protocols. Each code type will have a specific guard time, used to tackle issues such as interference from other messages and propagation delay, but most importantly allowing potential reflections of the pulse to break down. In this modulation scheme, a byte is encoded as two pulses within two windows of 16 time slots each. Each window represents 4 bits, one of them left-shifted by 4. Each slot in a window is of predefined length, and each slot represents a lookup in a binary table, such as the one given in table 2.1. Two windows together will be equal to a value between 0 and 255 (1 byte). To send more information, more pairs of windows like this can be appended to the message, keeping the code specific guard time in-between. At the end of the message, a checksum of some form should be provided, coded in the same manner.

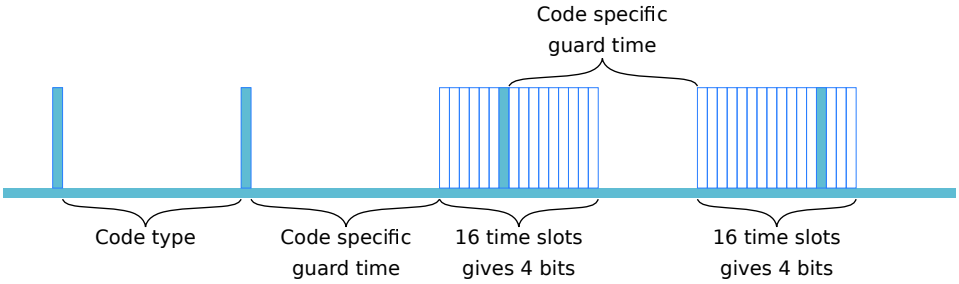


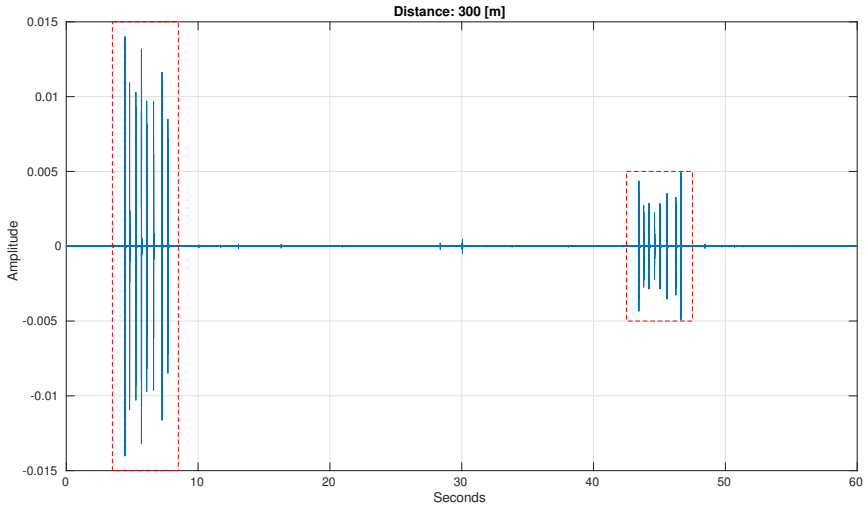
Figure 2.1: Differential pulse modulation scheme example. The time between the first two pulses represents code type. The code type determines the length of the guard time. Following that are windows of 16 time slots in pairs. Each window represents 4 bits, and combining two provides 1 byte of information.

Here a full example on how to use and decode the coding scheme is presented, based on fig. 2.1 and table 2.1. There is 360 ms between the first two pulses, so it is code type S256. In context of acoustic tags, S256 is used to send 1 byte tag id and 1 byte tag data. The code specific guard time is 360 ms, and each time slot of a window is 20 ms. Had fig. 2.1 been

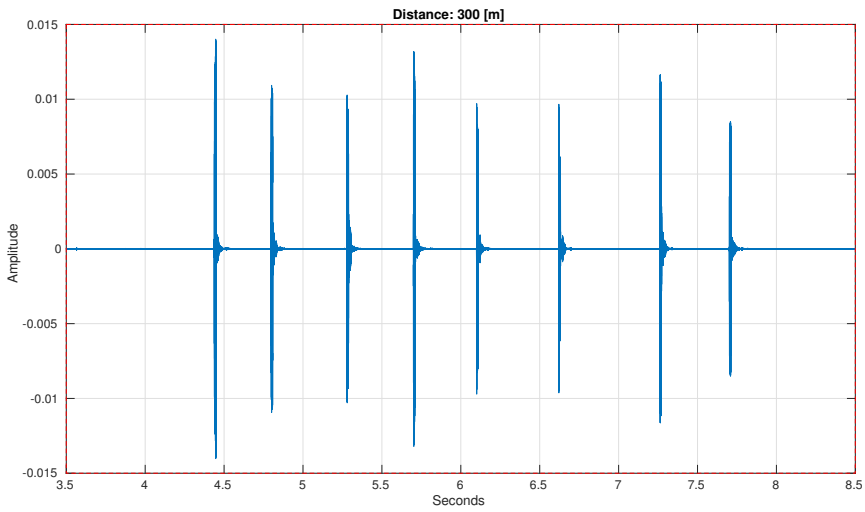
a full S256 message, 8 pulses would be shown: 2 for codetype and guard time, 2 for tag id, 2 for tag data, and 2 for checksum. Here only 1 byte of information is included. Assuming the first window is the most significant bits (MSB), it will be equal to 96, since it is in the 7th time slot (140 ms) of the window. Notice that the guard time starts immediately after the pulse. This is for shorter total transmission time, and is also why messages will have varying message lengths. In the second window, the pulse is in the 13th time slot (260 ms), i.e. the least significant bits (LSB) are equal to 12. Combining them, it can be determined that the byte is equal to 06 + 12 = 108. Figure 2.2 shows actual recordings of S256 messages using this scheme of pulse modulation. Noise is more dominant further away from the tag, making decoding more difficult.

Table 2.1: Example of binary table for decoding two 4-bit numbers as one byte in a pulse modulated scheme

Slot	Time	Binary	LSB	MSB
1	20 ms	0 0 0 0	0	0
2	40 ms	0 0 0 1	1	16
3	60 ms	0 0 1 0	2	32
4	80 ms	0 0 1 1	3	48
5	100 ms	0 1 0 0	4	64
6	120 ms	0 1 0 1	5	80
7	140 ms	0 1 1 0	6	96
8	160 ms	0 1 1 1	7	112
9	180 ms	1 0 0 0	8	128
10	200 ms	1 0 0 1	9	144
11	220 ms	1 0 1 0	10	160
12	240 ms	1 0 1 1	11	176
13	260 ms	1 1 0 0	12	192
14	280 ms	1 1 0 1	13	208
15	300 ms	1 1 1 0	14	224
16	320 ms	1 1 1 1	15	240

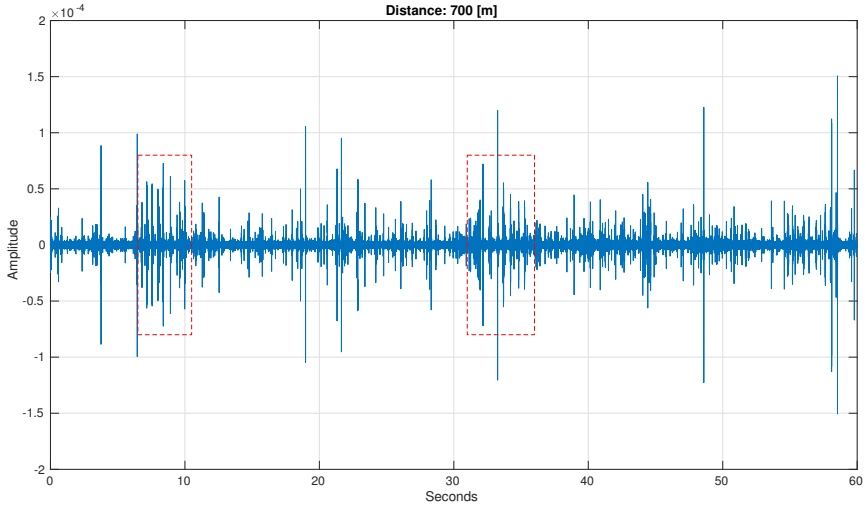


(a)

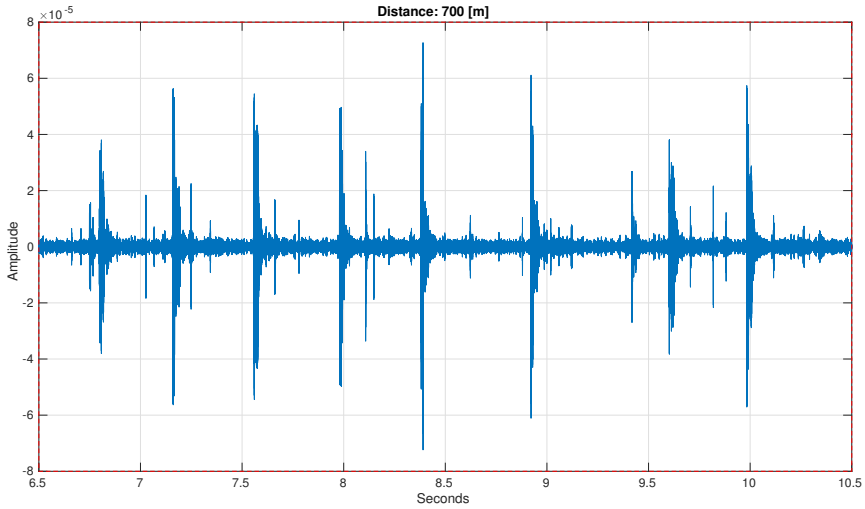


(b)

Figure 2.2: Audio data from a digital hydrophone filtered with a 10^{th} order Butterworth filter around 67 000 Hz with ± 500 Hz width. The 8 spikes represent the pulses, the time between them the data of the message. The sound detected originates from an acoustic tag sending data with the S256 protocol. **(a)** Shows a minute of audio recording containing two messages transmitted 300 meters away. **(b)** shows a zoomed-in view of the first message in **(a)**. **(c)** Shows a minute of audio recording containing two messages transmitted 700 meters away. **(d)** Shows a zoomed-in view of the first message in **(c)**



(c)



(d)

Figure 2.2: Audio data from a digital hydrophone filtered with a 10^{th} order Butterworth filter around 67 000 Hz with ± 500 Hz width. The 8 spikes represent the pulses, the time between them the data of the message. The sound detected originates from an acoustic tag sending data with the S256 protocol. **(a)** Shows a minute of audio recording containing two messages transmitted 300 meters away. **(b)** shows a zoomed-in view of the first message in **(a)**. **(c)** Shows a minute of audio recording containing two messages transmitted 700 meters away. **(d)** Shows a zoomed-in view of the first message in **(c)**. **(cont.)**

2.1.4 Advantages and limitations of fish telemetry

A multitude of sensors can be used in the tags for fish observation: Temperature, depth, pressure, conductivity, movement, heartbeats, and more. To have this breadth of data types available for monitoring individual fish is clearly valuable. Other observation methods are mostly limited to extrapolation of individual behavioural data from populations or groups of fish, and cannot provide physiological data directly. In addition, acoustic telemetry works over long distances, providing continuous data. In contrast, other technologies such as underwater camera systems, are limited to specific locations, only observing groups of fish that happen to swim by. Also, positioning of the fish can be done by having at least 3 hydrophones around the sea cage (section 2.2), a viable option for commercial sea cages.

Fish telemetry, however, has some limitations in addition to the ones already mentioned in this chapter. Data collected from acoustic tags can never be representative of a complete sea cage population. Sea cages often has populations consisting of 150 000 - 200 000 individuals, requiring manual surgery of large number of fish to be representative. Thus it requires high costs in manpower, equipment and time, while also increasing risk of reduced fish welfare.

The amount of tags possible to use simultaneously is limited, as the sound environment must be clear for minimal message collision. A single message can be 3-4 seconds in length, requiring minimal noise within that time-frame. As the number of tags increase, so does the risk of interference from other tags when detecting transmissions. Tags are typically coded to transmit data at pseudo-random intervals, such as once every 20-40 seconds, or once every 60-90 seconds, in a specific frequency. This minimizes collision danger, but data loss to some degree is inevitable. By using hydrophones that is able to listen to multiple frequencies simultaneously, the number of tags can be increased.

Tags are expensive considering they are designed for a limited lifetime. Recovering tags after use is challenging as the fish can die naturally during its lifetime, escape from the sea cage, be taken out manually without farmers noticing marks of a tag, or it can simply never be discovered during slaughtering.

2.2 Positioning of fish

A use case for the system developed in this paper is positioning of fish. Specifically, implementing 3D plotting of fish positions in the end-user web app tool. This section outlines the background theory of how positioning can be done with acoustic tags in sea cages, before more details on the underlying mathematics is presented in. In chapter 4, positioning algorithm pseudocode and considerations are presented, and in chapter 6 results from project use-case is presented. For more details on the theory behind these principles, or experiments done with them, see [10] and [24].

2.2.1 Principle of positioning of fish

There exists several positioning techniques that can be used to locate where a message is transmitted from. One technique is trilateration, where a transmitter and receiver are synchronized in time, and so the distance between them can be measured by comparing the time of arrival of the signal. Using this technique, it is possible to locate the source as a point on a circle (in 3D it will be a sphere). Using only one receiver will not be accurate enough, as the source can be on any point of that circle. Having two receivers, the possible locations are limited to only two locations. Adding a third receiver, one of the solutions can be eliminated, and the point where all three circles (spheres for 3D) intersect is determined as the original transmission location.

However, this cannot be used in situations where it is not viable to have synchronized time between receiver and transmitter, or where two-way communication is simply not possible, or desirable. Hyperbolic positioning, also known as time difference of arrival (TDOA) positioning, does not require synchronization of the transmitter and receivers. TDOA requires the location of each receiver to be known, and that the receivers clocks are accurately synchronized with each other. Rather than knowing when the source was transmitted, and looking at time of arrival, TDOA is looking at the difference between arrival times. In two dimensions, the difference of arrival between two receivers will provide a hyperbolic curve. Adding another receiver will give another hyperbolic curve. Ideally, solving the hyperbolic curves will provide a single intersection between them, less ideally two positions, and worst case no solution. This can be extended to three dimensions,

but the hyperbola will then be hyperboloids, meaning a minimum of 4 receivers are needed to give a single 3D position as a solution. In the general case, at least one more receiver is needed than the number of coordinates desired.

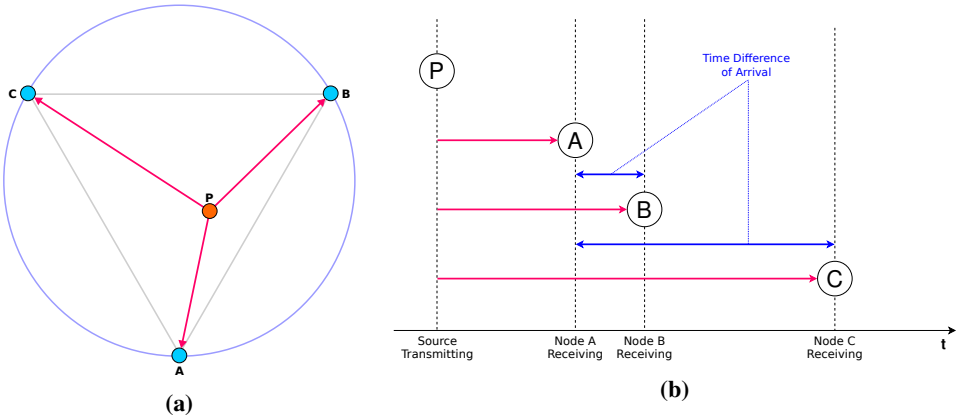


Figure 2.3: Principle behind TDOA. P is the location of a tag that is transmitting a message, while A, B, and C are the locations of hydrophones listening. TDOA requires known location of A, B, and C, as well as synchronized times between them. **(a)** The pink arrows represents the length of time before reaching the hydrophone. Blue circle going through the hydrophones represents a commercial sea cage setting. The gray triangle represents an area of high accuracy when localizing transmitter source. **(b)** Showing how the time difference of arrival is determined. It is these two differences that are used, between A, B and A, C, regardless of which hydrophone received it first. The locations are used to create a local coordinate system, so one should order the hydrophones the same each time, at least for the cage TDOA setting, to have compatible coordinates.

In the context of a sea cage and acoustic telemetry, it is possible to use hyperbolic positioning to locate the location of a tag. Assuming a situation where at least 3 hydrophones are on the same level vertically, a two dimensional solution can be found. This will give the horizontal coordinates of the tag. The acoustic tag can transmit the third dimension, for example using a pressure sensor to transmit depth, which is linearly correlated to pressure, thus eliminating the need for a 4th receiver. A device is required to synchronize the clock of each hydrophone, for example GPS. Also, how they are positioned in relation to each other matters. Figure 2.3 shows a setup of a sea cage, three hydrophones, and a tag transmitting a message. Any message originating within the triangle in fig. 2.3 will have the highest accuracy. One exception is the area in close proximity to the receivers, but there exists a potential solution for this case (section 2.2.2). Outside the triangle, more dilution of precision will happen, reducing the accuracy of potential solutions. In principle, this

problem stems from a large area of potential solutions (section 2.2.4). If receivers are placed relatively close to each other, and the time difference of arrival is relatively small, this will give a large area of potential solutions. An optimal setup would therefore be a setup where the sea cage is contained within a equilateral triangle spanned out by the hydrophones. Together with IoT-technology, this allows fast real-time 3D tracking of fish given fast enough message transmissions from the tag and processing power in the end receiver.

2.2.2 Details on TDOA positioning for fish

Given three hydrophones A, B, and C with known positions and synchronized clocks, and an acoustic tag transmitting its own depth, we can set-up the necessary equations to determine tag position fix. Assuming they share the same depth z_c, the hydrophones will span out a xy-plane. In this plane, we can set A as the reference receiver, located in $(0,\ 0)$. Defining the distance between A and B as b, we can place B on the x-axis $(b,\ 0)$. Finally, the distance between A and C can be defined as $c = \sqrt{c_x^2 + c_y^2}$, so that C has coordinates $(c_x,\ c_y)$. This will give us a local coordinate system for positioning, making the following equations easier to work with. Next, we need to define time of arrival of tag signal at location A, B, and C. Defining these as T_a, T_b, and T_c, we can further define difference of time of arrival in B and C, compared to A, as $T_{ab} = T_a - T_b$ and $T_{ac} = T_a - T_c$. Underwater, our signal transmission medium, is commonly assumed to have a constant speed of sound V of 1500 m/s. Multiplying time difference of arrival with sound speed, we have converted measured difference in time of arrival to range difference R_{ab} and R_{ac} from the transmission position to the stations.

Two stationary points in a plane will be the foci points of a hyperbola (hyperboloid in three dimensions). A hyperbola is a set of points such that for any points of the set, the absolute difference of distance between the points and the two stationary points, is constant. We have three stationary hydrophones where the difference in distance to all possible tag locations will be constant. Furthermore, our range differences, time of arrival [s] multiplied with signal velocity [m/s], is the distance between tag location and stationary

hydrophones. With this in mind, we can define the following equations:

$$\sqrt{x^2 + y^2 + z^2} - \sqrt{(x-b)^2 + y^2 + z^2} = V \cdot T_{ab} = R_{ab} \tag{2.1}$$

$$\sqrt{x^2 + y^2 + z^2} - \sqrt{(x-c_x)^2 + (y-c_y)^2 + z^2} = V \cdot T_{ac} = R_{ac} \tag{2.2}$$

Transposing, squaring and simplifying eqs. (2.1) and (2.2), we can write them as:

$$R_{ab}^2 - b^2 + 2bx = 2R_{ab}\sqrt{x^2 + y^2 + z^2} \tag{2.3}$$

$$R_{ac}^2 - c^2 + 2c_x x + 2c_y y = 2R_{ac}\sqrt{x^2 + y^2 + z^2} \tag{2.4}$$

Bertrand T. Fang [10] demonstrated that a solution for eqs. (2.1) and (2.2) can be obtained with eqs. (2.5) to (2.13):

$$y = gx + h \tag{2.5}$$

Substituting eq. (2.5) into eq. (2.3), one obtains:

$$z = \pm\sqrt{dx^2 + ex + f} \tag{2.6}$$

$$z^2 = dx^2 + ex + f \tag{2.7}$$

where

$$g = \left(R_{ac}\frac{b}{R_{ab}} - c_x\right)/c_y \tag{2.8}$$

$$h = \left(c^2 - R_{ac}^2 + R_{ac}R_{ab}\left(1 - \frac{b}{R_{ab}}^2\right)\right)/2c_y \tag{2.9}$$

$$d = -\left(1 - \frac{b}{R_{ab}}^2 + g^2\right) \tag{2.10}$$

$$e = b\left(1 - \frac{b}{R_{ab}}^2\right) - 2gh \tag{2.11}$$

$$f = \frac{R_{ab}^2}{4}\left(1 - \frac{b}{R_{ab}}^2\right) - h^2 \tag{2.12}$$

Finally, we can define the tag position vector, depending on a single unknown parameter

x, as follows:

$$\vec{R} = x\vec{i} + (gx + h)\vec{j} \pm \sqrt{dx^2 + ex + f}\vec{k} \tag{2.13}$$

With these polynomial equations, we will achieve two candidate points representing possible locations for the transmission source. There are some special cases to be aware of. If $R_{ab} = 0$, eqs. (2.8) to (2.12) will contain division by zero. However, geometrically, $R_{ab} = 0$ implies $x = b/2$, and so we can solve eq. (2.13) by expressing it in terms of y instead of x. Similarly, when $R_{ac} = 0$, the position will be on the line perpendicular to the midpoint between A and C, i.e. $y = -c_x/c_y \cdot x + c^2/(2c_y)$. However, there is no need to handle this case separately, as there is no division of R_{ac}. In the case where $R_{ab} = R_{ac} = 0$, the solution is trivial, as the position will be equally distant to all three stations, i.e. $y = (c^2 - bc_x)/2c_y$. The only other special case to handle when solving the equations is therefore when $R_{ab} = 0$ and $R_{ac} \neq 0$. We already know that $x = b/2$ in this case, so all that remains is to find the y-candidates. If we square and transpose eq. (2.4), putting it on quadratic form, we get:

$$a_2 y^2 + a_1 y + a_0 = 0 \tag{2.14}$$

where

$$a_2 = 4\left(c_y^2 - R_{ac}^2\right) \tag{2.15}$$

$$a_1 = 4c_y\left(R_{ac}^2 - c^2 + 2c_x x\right) \tag{2.16}$$

$$a_0 = \left(R_{ac}^2 - c^2 + 2c_x x\right)^2 - 4R_{ac}^2\left(x^2 + z^2\right) \tag{2.17}$$

The roots of the polynomial in eq. (2.14) will be the y-candidates of the position in this case.

2.2.3 Resolving position fix

There are in principle two ways to determine position candidate ambiguity. First, to use knowledge of which hydrophone received the message first. Second, to use a radius from the sea cage center to which you limit valid position fixes to. In this section, it is assumed

that the hydrophones are located in the same plane for an easier 2-dimensional argument. The same argument will hold for three dimensions.

Given a sea cage with three hydrophones A, B, and C with synchronized clocks placed on its circumference. We know time of arrival of each hydrophone, T_a, T_b, and T_c, and we know their locations, $P_A = (A_x,\ A_y)$, $P_B = (B_x,\ B_y)$, and $P_C = (C_x,\ C_y)$. With two position candidates, $P_0 = (x_0,\ y_0)$ and $P_1 = (x_1,\ y_1)$, we can calculate their distance from the receivers. These distances can be defined as:

$$d_{N_i} = \sqrt{(x_i - N_x)^2 + (y_i - N_y)^2} \tag{2.18}$$

where $N = A,\ B,\ C$ and $i = 0,\ 1$ (P_0, P_1).

Let us say that hydrophone A received the tag message first, C received it last: $T_a < T_b < T_c$. This means that if position P_0 is correct, then $d_{A_0} < d_{B_0}$ should be true since $T_a < T_b$, but at the same time, $d_{B_1} < d_{A_1}$ must be true to determine it unambiguously. This is a good starting point for determining position, but what if $T_a = T_b$? We still know that $T_a < T_c$, and so we can follow the same argument for this pair of hydrophones. Sorting the time of arrival of each hydrophone in ascending order, we get: T_F, T_S, T_T, and consequently: d_{F_0}, d_{S_0}, d_{T_0}, d_{F_1}, d_{S_1}, d_{T_1} ($_F = First$, $_S = Second$, $_T = Third$). Using this, we get the procedure defined in algorithm 1.

RESOLVE-POSITION-BASED-ON-TOA(d, P_0, P_1)

```
1  // d contains distances between P0, P1, and _F, _S, and _T
2  if d.F0 < d.S0 and d.S1 < d.F1
3      position = P0
4  elseif d.F1 < d.S1 and d.S0 < d.F0
5      position = P1
6  elseif d.S0 < d.T0 and d.T1 < d.S1
7      position = P0
8  elseif d.S1 < d.T1 and d.T0 < d.S0
9      position = P1
10 else
11     position = None                  // Further resolving of position needed
```

Algorithm 1: Time of Arrival based resolving of position ambiguity

If we are still unable to determine the correct transmission origin, we can further try to resolve the position fix with a sea cage radius argument. The cage has center $cg = (cg_x,\ cg_y)$ and radius r. The position candidates have distances d_{cg_0} and d_{cg_1} from the center. With this we can use the procedure outlined in algorithm 2 to hopefully unambiguously determine a position candidate.

```
VERIFY-POSITION-WITHIN-SEA-CAGE(position, P_0, P_1)
 1  THRESHOLD = 1.1 // or any other value
 2  r_max = r · THRESHOLD
 3  if position ≠ NONE        // Check if a position candidate has already been chosen
 4      d_cg = distance(position, cg)
 5      if d_cg ≥ r_max
 6          position = NONE                          // Chosen position is invalid
 7  else // Resolve ambiguity with distance to cage center
 8      if d_cg0 < r_max or d_cg1 < r_max
 9          if d_cg0 < d_cg1
10              position = P_0
11          else
12              position = P_1
13      else
14          position = NONE                          // No position fix could be found
```

Algorithm 2: Sea cage radius based resolving of position ambiguity

In section 2.2.1 it was mentioned that it is difficult to find position fixes in close proximity to the receiver locations. However, the signal strength for that particular receiver will be very strong due to the short distance, and it should be stronger than the signal strength of the other two receivers by a good margin. So if no position fixes can be found, but the signal strength at one receiver is great and much larger than the other two, one could set the position to be at the location of the receiver, or a short distance away from the receiver.

2.2.4 Positioning Dilution of Precision

The accuracy of TDOA if affected by several factors, such as relative geometry of receivers, and timestamp accuracy. By placing the hydrophone receivers in a equilateral triangle

around a sea cage, minimal distortion due to geometry can be achieved. If the receiver clocks support 1 millisecond resolution, the upper bound of accuracy will be 1.5 m since sound speed underwater is assumed constant 1500 m/s. This should be sufficient for the use case of sea cages. Dilution of precision (DOP), also called geometric dilution of precision (GDOP), describe errors caused by the relative position of GPS satellites. This can be expressed as eq. (2.19) [8].

$$GDOP = \frac{\Delta(\text{Output Location})}{\Delta(\text{Measured Data})} \tag{2.19}$$

Δ(Measured Data) is term to describe changing errors on a measurements. So, in ideal situations, small changes will not lead to large changes in Δ(Output Location). There are different types of DOP, like VDOP (vertical), PDOP (position 3D), TDOP (time) and HDOP (horizontal). The `MATLAB` code provided in appendix B of [9] simulates HDOP effect of a sea cage setting. The code provided was translated to Python, and similar simulation visualization was produced to give section 2.2.4. Here the effect of receiver geometry is clearly seen. The triangle of the visualization show optimal HDOP for the cage. One can see how the position accuracy quickly deteriorates beyond the triangle area. For this simulation, the transmission source locations are placed in the same plane as the receivers.

A challenge of positioning of fish using underwater acoustic signals, is the complexity involved. Acoustics produce a very complex environment with different speeds depending on depth, reflections of object such as the sea floor, and the sound waves propagating through water bend in certain paths. The simplification normally used is that sound moves in a straight line with constant sound speed, but this is not accurate for the truly complex acoustic picture underwater. Additionally, the signal a tag sends out is a spherical energy signal produced by a small transmitter. The hydrophone in turn detects a very small pinpoint of the sphere, meaning a very small amount of energy. How does the hydrophone determine time of arrival? Exactly when the energy started to hit the sensor? Directly after the signal has passed? In this regard, determining good ways to perform signal processing (FFT) is a challenge. This too affects the resulting positioning of fish. Positioning is in other words not a trivial matter, despite a relatively simple analytical solution. It has many

challenges, and several sources of error. Despite this, it is still a highly valuable insight that provides insight into how fish behaves. Perhaps not with pinpoint accuracy, but with accuracy good enough to see generally which part of a sea cage, and which depth, a fish is in.

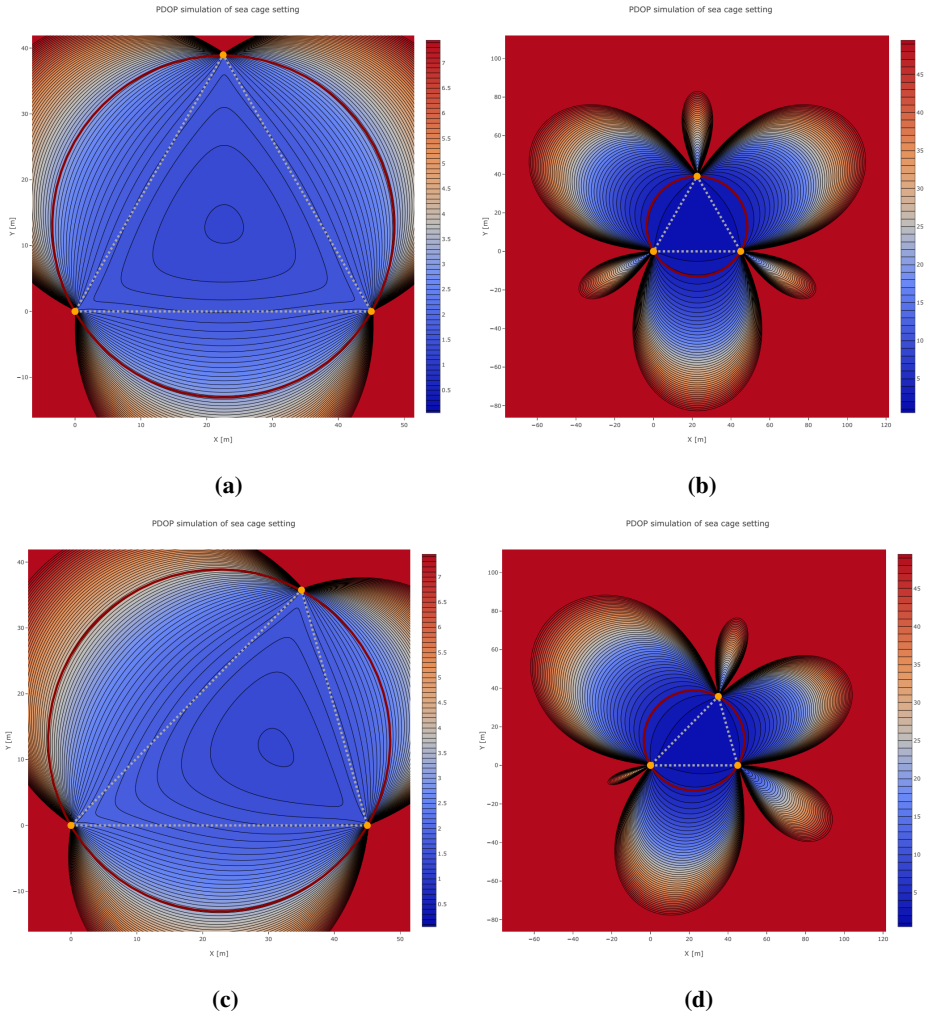


Figure 2.4: PDOP simulation in cage setting. **(a)** shows an ideal setup with a equilateral triangle, in this case with a threshold set to 7.5 meters. **(b)** shows a zoomed out version of **(a)**, but with threshold set to 50. **(c)** shows a non-ideal setup with threshold set to 7.5. **(d)** shows a zoomed out version of **(c)**, but threshold set to 50.

2.3 Communication protocols and connectivity

In this section, some definitions for communication protocols used in the system are defined. In chapter 4, these definitions will be used to explain the design of the system.

2.3.1 Internet of things (IoT) definition

There are many different ways to define IoT, and a simple definition to cover all ways the term can be interpreted is not an easy task. Different definitions can be defined depending on whether the system in question is a global system, bound to a region, or a small local system. Whether it is controlling actuators in an interconnected network of things, or only forwarding data. However, there are some definitions more prominent than others. For a small environment scenario, the following definition is an apt description:

Internet of things (IoT) *An IoT is a network that connects uniquely identifiable "Things" to the Internet. The "Things" have sensing/actuation and potential programmability capabilities. Through the exploitation of unique identification and sensing, information about the "Thing" can be collected and the state of the 'Thing' can be changed from anywhere, anytime, by anything* [25].

2.3.2 MQTT connectivity protocol

The client server publish/subscribe format is well-suited for IoT-applications. An IoT-device can connect to a server and subscribe to a topic. It can then publish messages to the server using this topic, as well as listen for messages on topics it subscribes to. The server will forward any messages it receives on a topic to all clients subscribing to that topic. This means that several users can use the data decoupled from each other. MQTT is a protocol based on this format, well-suited and often used in IoT devices:

MQTT - *MQTT is a Client Server publish/subscribe messaging transport protocol. It is light weight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet*

of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium [4].

2.3.3 LoRaWAN

Communication is important for IoT-devices, as they are designed to be connected. Many devices, such as those used in so-called smart homes, can simply connect to a local wireless LAN that provides internet. However, many devices are meant to be used far away from internet infrastructure, so intermediate links are needed for the data to flow to the internet. LoRaWAN is such a link, well-suited for IoT:

LoRaWAN - *The LoRaWAN specification is a Low Power, Wide Area (LPWA) networking protocol designed to wirelessly connect battery operated 'things' to the internet in regional, national or global networks, and targets key Internet of Things (IoT) requirements such as bi-directional communication, end-to-end security, mobility and localization services* [1].

Chapter 3

Requirements

This chapter presents the requirements of the IoF concept in the different network architecture layers. The complete network architecture is described in section 3.1 together with a high-level design of the system from the perception layer to the application layer. In sections 3.2 to 3.4 the requirements of the system components in each layer of the architecture is identified and presented. For the application layer, detailed requirements are defined in the form of functional requirements (FR), nonfunctional requirements (NFR), and UI requirements (UIR).

3.1 Network architecture

A natural network architecture abstraction for an IoT system is to divide it into three layers: perception, network and application. In turn, the same division is natural for IoF. Figure 3.1 presents a high-level design of the IoF concept. The arrow heads indicate which way data is transferred. Technically, several of the links could support two-way communication, like the gateway and MQTT server, but in practice this communication path is unused, as the perception layer only serves to push data to the application layer. In future implementations of the IoF concept, this option could be utilized to update perception layer firmware, or to control operations with actuators. However, this is some way into the future, until the interpretation, decision, and act phases of PFF is more developed industry-wide, and a

closed feedback-loop with acoustic telemetry is possible.

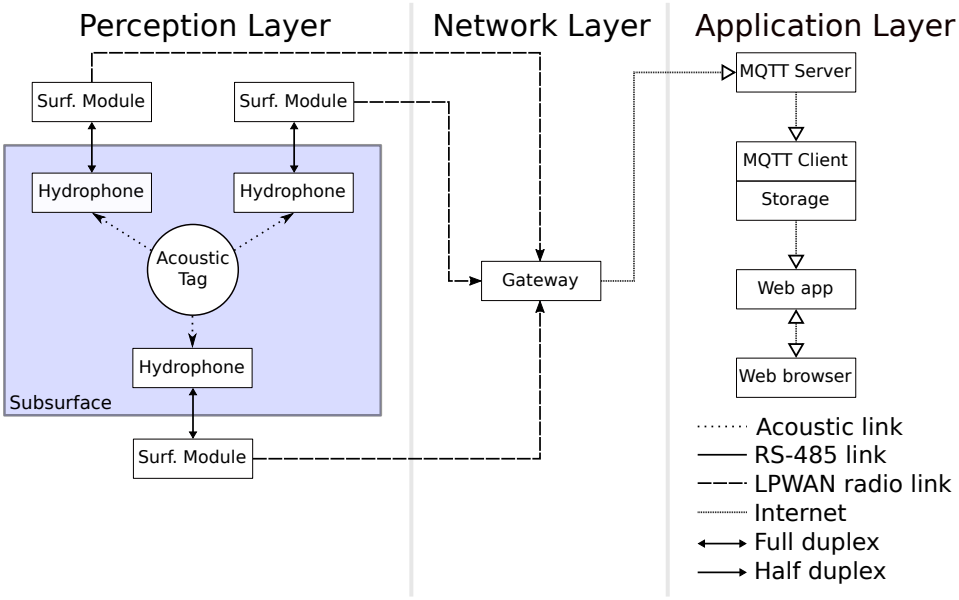


Figure 3.1: Network architecture of IoF concept. Full duplex represents two-way communication, while half duplex is only one-way. Technically, some of the half duplex links support two-way communication, but in this case direction of data flow is prioritized, as the perception layer and network layer only serve to push data to the application layer. Web browser represents end-user(s).

3.2 Perception layer

3.2.1 Acoustic tag requirements

Acoustic tags need to reliably and accurately transmit desired data types in chosen pseudo-random intervals for a designated operation time. The most important task in relation to the IoF concept is to identify characteristics, i.e. which type of fish, what data, operation time, etc. After identifying the appropriate setup, tags with characteristics close to these parameters can be ordered from a company producing acoustic tags.

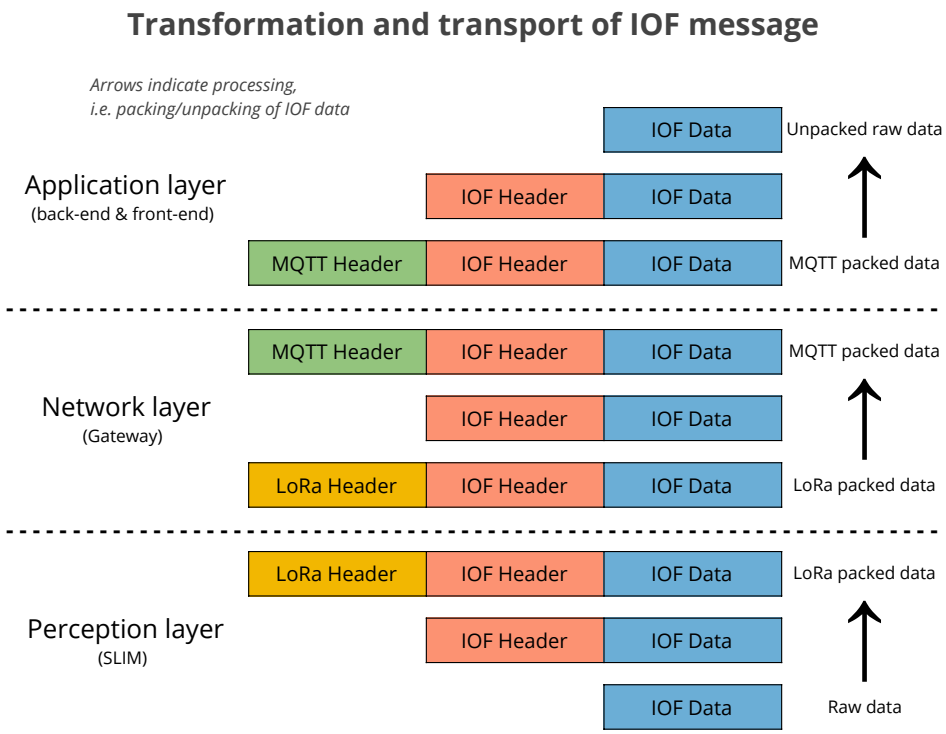


Figure 3.2: IoF message transport through network architecture with corresponding transformation.

3.2.2 Hydrophone requirements

For positioning, at least three hydrophones are required. Having another hydrophone could be good for redundancy, but it is not required. The hydrophones needs to be able to listen for tag messages on at least one frequency, preferably more. It needs to decode tag messages, and support two-way communication to the synchronization and LoRa interface module through RS-485. Naturally, the hydrophone should be designed for reliable use underwater over longer periods of time, but similarly to the tags, there exists many available hydrophone solutions from a multitude of companies.

3.2.3 Synchronization and LoRa Interface Modules requirements

The synchronization and LoRa interface module (SLIM) needs accurate GPS to synchronize the clock of the hydrophone it is connected to. This is as mentioned in section 2.2 essential

in positioning of fish. In sea cage settings, the hydrophones are likely to be stationary, so a one-time accurate GPS read of the SLIM / hydrophone location can be done. However, an added benefit of GPS in the SLIM is that it can report its own position, which is especially useful for wild fish use-cases. SLIM also needs a LoRaWAN interface, to transmit data to a nearby internet-connected gateway, and a RS-485 interface to communicate with the attached hydrophone. The SLIM must be able to read TBR data and pack it into a predefine data protocol, which it in turn forwards through the LoRaWAN interface. Besides this, the module needs to be power efficient to maximize run-time, and it needs to be physically robust to withstand difficult environments. Effectively compressing data is a trade-off between more processing and less LoRa transmission. Compressing will require more power, but transmitting less will reduce power usage, and will also allow more data to be transferred before reaching LoRa limitations.

3.2.4 LoRaWAN requirements

LoRaWAN has a couple of rules built-in determining the legal limit of broadcasting data when using the LoRaWAN network. These rules determine the frequency channels and data rates to be used, where the data rate of a node is a trade-off between range and duration of messages. This means that there are limitations in how much data a LoRaWAN node is allowed to send per day. This clearly creates a need for compression of data, to maximize amount of data a node can send. Therefore the design of the IoF message packets is an important requirement. With the configuration that is intended to be used for the IoF concept, a maximum message size of 128 bytes can be attained, sent each minute.

3.3 Network layer

3.3.1 Gateway

A gateway is required to unpack the LoRa messages from SLIMs, pack them to MQTT, and send to the MQTT server. The gateway requires power and a solid internet connection, either in the form of a wall socket and internet modem, or as batteries and a mobile network sim-card. The latter is less susceptible to power outages and bad internet infrastructure at

remote locations, and should be preferred in situations where mobile network connection is available. The gateway needs to be able to send the messages on appropriate MQTT topics, either a topic for each ID in the local project (can be the hydrophone serial numbers for example), or following a predefined topic configuration.

3.4 Application layer

3.4.1 Back-end requirements

To explain the requirements of the back-end, it is useful to first follow the data flow of the whole system: (1) An acoustic tag message is detected in a hydrophone, and it is sent over RS-485 to a synchronization and LoRa module (SLIM). (2) The SLIM periodically compress new tag detections and hydrophone sensor readings into LoRa-messages which it broadcasts to a gateway on-site. (3) The gateway unpacks the LoRa-message, packs it into the MQTT message format, and forwards it to a MQTT server on a predefined topic. (4) The server will then forward the MQTT message to any appropriate subscribing MQTT clients.

This is where the back-end enters. The first requirement of the back-end will therefore be to function as a MQTT client. By subscribing to the wildcard # topic, every message the server receives will be forwarded to the MQTT client. However, the main advantage of a MQTT server is that one server can be used across multiple projects. By subscribing to the # topic, mixing of different projects will occur. So it is advisable to have each back-end client subscribe to specific project topics. Next, the back-end should successfully decode the bytes of the MQTT message into readable data, following a predefined data protocol. Consequentially, both the SLIM and back-end client needs to know the same data protocol. While doing so, the back-end should also be able to perform tag or project specific formatting of data, like converting bytes data to depth data. The second requirement of the back-end is therefore to unpack and convert incoming messages. After successfully unpacking the data, it should be converted to a reasonable format and stored in persistent storage. This is the third requirement of the back-end. Lastly, the back-end should be able to perform positioning of all triplets of tag detections received, and store the results to

persistent storage. By having MQTT as the entry-point, and some sort of storage as output, the back-end will have modularity built-in.

These are the requirements the back-end absolutely must meet. Any additional requirements are for better performance, expected behaviour, or desirable behaviour. The most important requirements are listed again as functional requirements (FR) below, while additional requirements for back-end are listed as non-functional requirements (NFR). FR requirements are generally focused on what the software must do, while NFR requirements focus more on how it should do it. Finally, any user interface requirements are listed as UI requirements (UIR). UIR requirements are functional requirements.

Functional back-end requirements (FRs)

- **FR1 - MQTT client:** able to handle and serve incoming MQTT messages.
  - **FR1.1:** Able to subscribe to different project-specific MQTT topics.
  - **FR1.2:** Support user/password authentication and ssl.
  - **FR1.3:** Support MQTT version 3.1.1 or newer.
- **FR2 - IoF message parsing:** able to take IoF bytes messages as input and parse them to retrieve IoF data.
  - **FR2.1:** Use predefined protocol format to unpack.
  - **FR2.2:** Able to convert raw data to appropriate formats while unpacking.
- **FR3 - Store IoF data to persistent storage:** store raw, converted, and positional data to storage.
- **FR4 - Perform TDOA positioning:** solve TDOA algorithm to find positions for incoming IoF data.

Non-Functional back-end requirements (NFRs)

- **NFR1 - Fault tolerance:** restart back-end if it crashes (for example process pairs).
- **NFR2 - Redundancy:** store a backup of data received in case of data corruption, wrongful conversion of data, or other unpredictable events.
- **NFR3 - Persistent storage:** should be reasonable, flexible, and easily maintainable.

  - **NFR3.1:** Define structure and format of persistent IoF data.
  - **NFR3.2:** Database solutions such as SQL databases preferred over simple solutions like storing raw bytes in `.txt`-files, or niche file formats not widely adopted.

- **NFR4 - Error handling:** use of typical error handling methods (`try-except`).
  - **NFR4.1:** Log errors to file with included error metadata.
  - **NFR4.2:** Support different levels of errors, for example. 'warning' and 'critical'.
  - **NFR4.3:** Discard IoF data in case of corrupt or incorrect message structure, but store raw message in backup storage.

- **NFR5 - Able to apply metadata conversion:** which should be read from a file-system file.
  - **NFR5.1:** A standard format for metadata file structure.
  - **NFR5.2:** Storing data like temperature, depth, and acceleration directly in addition to raw data in persistent storage.

- **NFR6 - Perform positioning:** solve TDOA positioning for IoF system.
  - **NFR6.1:** Be able to use metadata to set positioning parameters such as TBR locations.
  - **NFR6.2:** Persistent storage of found positions in a local XYZ-coordinate system.
  - **NFR6.3:** Persistent storage of latitude-longitude coordinates (or other standard format) of found positions.
  - **NFR6.4:** Ability to find triplets in database fast and efficiently.
  - **NFR6.5:** Ability to find and adjust timestamps that have drifted fast and efficiently.
  - **NFR6.6:** Perform positioning on messages as they come in.
  - **NFR6.7:** Perform positioning on a complete database.
  - **NFR6.8:** Flexible back-end, allowing multiple ways to set-up positioning (metadata file, code context, source).

- **NFR7 - Convert raw hydrophone data:** to the IoF format, i.e. ability to input data logged locally in hydrophones to back-end to get IoF data format out.

UI back-end requirements (UIRs)

- **UIR1 -** GUI or command-line interface to setup MQTT connection.
- **UIR2 -** GUI or command-line interface to initialize new project.
- **UIR3 -** GUI or command-line interface to reset and/or delete current project.
- **UIR4 -** GUI or command-line interface to enable or disable varying levels of interface printing / error log writing.

3.4.2 Front-end requirements

The main general requirement for the front-end of IoF is to be accessible in a web browser without the need for end-users to install or configure anything, allowing them to interact with graphs that they can customize as they please, and to view data in real-time. It also needs to be responsive, i.e. not using a couple of minutes to plot large amounts of data. It should be project agnostic and require minimal configuration to work for new projects. To accomplish this, a standard format and procedure to configure and set-up projects needs to be defined.

These are the requirements the front-end should meet. Any additional requirements are for better performance, expected behaviour, or desirable behaviour. Especially front-end have many desired characteristics that can be defined in addition to the main requirements. The most important requirements are listed again as functional requirements (FR) below, while additional requirements for front-end are listed as non-functional requirements (NFR). Finally, any user interface requirements are listed as UI requirements (UIR).

Functional front-end requirements (FRs)

- **FR1 - Accessible through most web-browsers:** the application should be web-based, and be possible to access and use in common web browsers, without any additional installations in the end-user device.surface support modules

- **FR2 - Ability to view real-time data:** end-user should be able to graph plots that update in real-time.
- **FR3 - Responsive interface:** when clicking buttons and using menus, the response should be instantaneous, and when plotting graphs, it should be reasonably fast (< 30 seconds).
- **FR4 - Project agnostic:** The application should be easy to implement for different experiments, requiring minimal configuration and change in code and structure.
    - **FR4.1:** Documentation explaining how to set-up front-end should be written.
    - **FR4.2:** Standard format of metadata files, and procedure to set-up.
- **FR5 - Wide variety of plot types:** should be able to support several basic, scientific, statistical, and 3D charts.
    - **FR5.1:** Scatter, line, filled area plots.
    - **FR5.2:** Boxplots, histograms.
    - **FR5.3:** Contour plots and heatmaps.
    - **FR5.4:** 3D scatter, animation, and surface plots.
    - **FR5.5:** Timeseries plot
- **FR6 - Plot customization:** The user should be able to use basic controls to customize how the look is structured and its design.
    - **FR6.1:** Tools for zooming, panning, and selecting data etc.
    - **FR6.2:** Allowing the user to choose their own x-axis, y-axis, and/or z-axis.
    - **FR6.3:** Allowing the user to change styling of the plots, like axis labels, titles, data colors, marker and line sizes etc.

  surface support modules
- **FR7 - Data filtering:** giving user controls to filter the dataset(s) of the application to further customize desired visualization of data.

Non-Functional front-end requirements (NFRs)

- **NFR1 - Simple and flexible deployment of application:** easy set-up of webpage so that it can be accessed from anywhere.

- **NFR2 - Authentication:** at least username/password base, preferably an advanced and robust solution to hinder unwanted user access.
- **NFR3 - Table view:** of both filtered and raw data.
- **NFR4 - Data export:** of both filtered and raw data, and in multiple formats.
- **NFR5 - Export of plots:** in multiple formats and configurations.
  - **NFR5.1:** Formats such as pdf, svg, tiff, and png.
  - **NFR5.2:** Custom sizing of figures.
- **NFR6 - Allowing data upload:** to allow end-users to provide their own datasets for analysis.
  - **NFR6.1:** Beneficial if able to store uploaded data persistently, so that users can come back and continue their analysis without needing to re-upload.

UI front-end requirements (UIRs)

- **UIR1 - Dynamic webpage sizing:** adapting to the different devices end-user might want to use.
- **UIR2 - App gallery:** for separation of projects and/or applications, for example a separate application for real-time monitoring, and a separate application for table view and download of data.
- **UIR3 - Dashboard functionality:** for project management and potentially set-up of back-end too.
- **UIR4 - Webpage controls:** should be distinct, clearly labelled, and intuitive to use.
- **UIR5 - Navigation:** navigation between different data from IoF system should be implemented and easy to follow.

3.4.3 Shared requirements between front-end and back-end

Both the front-end and back-end share some desired traits and optimal behaviour, as they are software tools intended to work together. The persistent storage will be the end-point for back-end and the entry-point for front-end. Interfacing here should be modular, so that

any back-end and front-end configuration can work together. However, in practice, it would be beneficial for back-end to optimize storage for front-end, and communication between them could benefit performance of the overall system. While improving usefulness, it could make them completely dependent on each other. As development continues, this might make the whole system break functionality if tools of either underlying system break, and it would also limit the future of IoF to specific implementations, making the system less flexible.

Below are some shared desired characteristics, or non-functional requirements, of both back-end and front-end, that should be kept in mind when developing the IoF software:

- **NFR1 - Limit the code to Python -** and use a minimal amount of libraries and tools.
    - **NFR1.1:** Minimal number of dependencies on third-party libraries and tools. This helps to avoid breaking functionality as development progress.
    - **NFR1.2:** Small code footprint by limiting code to only python. The required knowledge and installation for developers and users become less complicated. Maintainability should consequentially be easier. It also enables easier communication between front-end and back-end, and less chance of library/tool updates breaking compatibility.
- **NFR2 - Use a new version of Python:** and keep in mind future development overhead when using tools and features.
    - **NFR2.1:** Avoid relying on modern tools that break or change functionality with each update.
    - **NFR2.2:** Avoid using tools/versions that will be obsolete in the near future.
- **NFR3 - Documentation of code:** preferably with a html/pdf-based companion.
    - **NFR3.1:** Should follow standard documentation practices, i.e. following a documentation style like the google style [14]). Good documentation improves maintenance, readability, and updateability. Any coding project benefits from good documentation, but it is especially important in projects where multiple people will end up working on the code, spread across years.

    - **NFR3.2:** Documentation in html-format/pdf-format as part of the code source, generated by a tool like sphinx [36]. This makes it even easier to configure and work with the code.

- **NFR4 - Able to operate independently from each other:** Front-end and back-end code not depending on runtime of the other)
- **NFR5 - Open-source based.**
    - **NFR5.1:** IoF code available in an open-source repository.
    - **NFR5.2:** IoF should use open source software, as open source software is free to use and generally well documented.
    - **NFR5.3:** Minimal use of proprietary libraries and tools. While they can be used, the benefits of both using and making code open-source is that feature requests can more easily be added, and help can be gained from tool developers and/or communities.

Chapter 4

Design and implementation

The description of the IoF concept has been in general terms, as the principles could easily be applied to several solutions in the perception layer. This is by intention to allow readers to form ideas on how to implement their own system. However, this chapter will discuss design choices and implementation, and so the focus will shift to what has been done for this specific implementation and thesis. The chapter is divided in the following sections:

- Section 4.1 explains the data types generated by the SLIM and defines the corresponding message formats and protocol.
- Section 4.2 covers the solution chosen for persistent storage with an accompanying discussion on the alternatives, and the benefits and disadvantages of them.
- Section 4.3 presents the materials and components used for the implementation, amongst them the Thelma BioTel TBR700 RT hydrophone (TBR for short).
- Section 4.4 gives details on the software tools and libraries used in both back-end and front-end.
- Section 4.5 presents how back-end and front-end was implemented. For back-end, the program flow, positioning, and fixing drifted timestamps. For front-end, Dash and MVC, typical dash app structure, the challenges of sharing data between callbacks, performance, and construction of the case-study apps. Figure 4.17 shows how the front-end webpage looked.

4.1 Message formats

As mentioned in chapter 3, compression of data is important to not exceed the limits of the LoRaWAN-network. Smaller messages means less bandwidth usage, which can quickly become a bottleneck if the amount of data sent from acoustic tags is higher than the available capacity. This was experienced first-hand during the case study in the previous implementation of IoF back-end. A number of messages were not sent because LoRaWAN capacity was reached.

4.1.1 Message data types and packets

First we have the tag detection packet. The acoustic tags produce data which they transmit to the TBRs. The data can be a multitude of things, depending on the used sensor(s), but it will most likely be in an unsigned byte format with some sort of compression. The burden of converting raw data should be left to the back-end, which is not battery-dependent. The tag data is transmitted with a certain codetype, an unsigned byte, which tells the TBR which frequency and which communication protocol scheme the tag is following. The TBR is constantly listening for signals, and determines a SNR value for each detection. The TBR also has a clock with millisecond precision, allowing it to store time of arrival. So in principle, the tag only produces data from its own sensors, which it transmits in some predefined manner on a predefined frequency, and then the TBR detects the transmission, store the data from it, and adds metadata to each detection. The added metadata is considered part of the tag data type. Together, these data types are grouped in a tag detection packet for the IoF protocol.

Second is the TBR sensor packet. The TBR has an internal temperature sensor, and as mentioned, it is constantly monitoring for new tag messages. This allows the TBR to measure ambient noise in the area, which it stores both as an average value and a peak value every predefined time period. The TBR does not use a communication scheme like the tags, but to distinguish the TBR messages from tag messages when transmitting to back-end, it is useful to give the TBR a distinct codetype value. Having the time of sensor reading as a timestamp is useful. There is however no need to have it in milliseconds, and so no millisecond value is needed. Lastly, the frequency in which the TBR listens for noise

can be included. The combination of these are considered as the TBR data type. Together, these data types are grouped in a TBR sensor packet for the IoF protocol.

Third is the SLIM packet. The SLIM supports GPS for clock synchronization of the TBR, and it can use the GPS to report its own position. The position is needed for positioning (section 2.2), and it could be especially useful in wild fish projects, or AUV situations [24]. A GPS reading includes several parameters, like PDOP, GPS fix, number of satellites it is tracking, and of course latitude and longitude values. All of these are useful for the IoF system, and they are considered to be the GPS data type. In addition to this, it would be useful for the end-user/site operators to know how much battery is left. Similar status updates from the SLIM module would also be good to include. Together, these data types are grouped in a SLIM status packet for the IoF protocol.

4.1.2 IoF Message frame

While there is only need for at most 1 TBR sensor packet and 1 SLIM packet in a IoF message, a packet of these types is only needed every 10th message or so. In contrast, there should, and can, be a multitude of tag detection packets in each IoF message. The SLIM is designed to broadcast a new message through LoRa every minute, and within one message there can theoretically be over 15 tag detections. So, when considering compression to keep bandwidth usage low, there is most to gain in compressing tag packets. The IoF message frame can be seen in fig. 4.1, and it is coded with the big endian byte format, i.e. the most significant bits come first. The difference between big and little endian can be seen in fig. 4.3. The header of the IoF message consists of 6 bytes where the first 2 bytes contains 14-bits dedicated to the TBR serial number, and 2 bits for a header flag. The remaining 4 bytes are for a reference `UNIX UTC` timestamp. Figure 4.2 shows the full design of the message header.

4.1.3 Tag detection packet

The first byte of the tag detection packet is a byte that gives the number of seconds passed since the reference timestamp set in header. If this is the packet used to set the reference time, it will always be equal to 0. The next byte is the most important one,

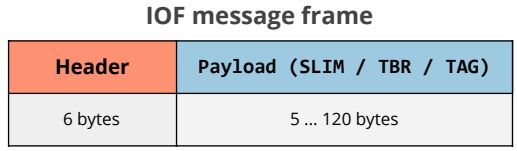


Figure 4.1: IoF message frame. Typically each tag packet will be 6 bytes. TBR and SLIM packets will only be sent once every predefined time interval. TBR packets can be unpacked with a codetype field shared with tag packets. SLIM packets will be first in the payload if the header flag has been set to 1.

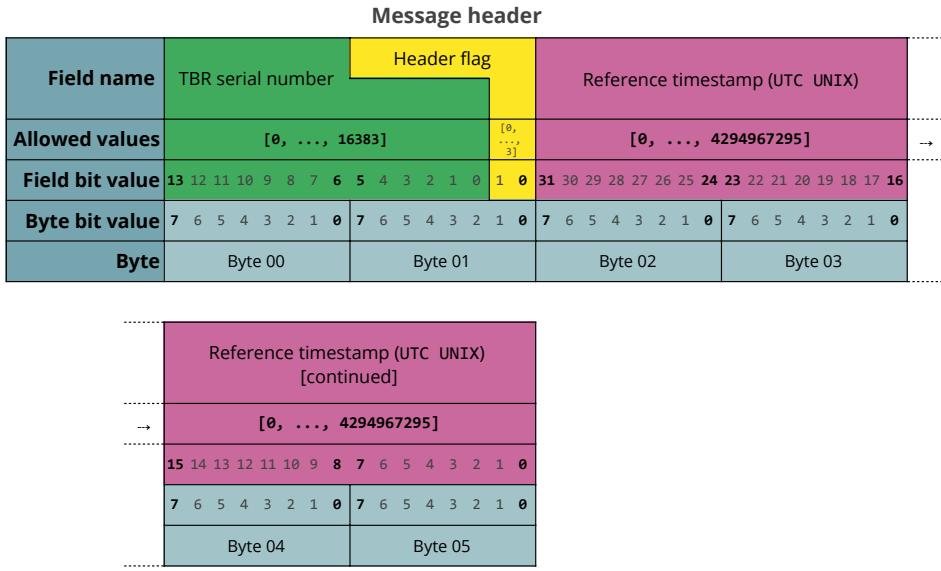


Figure 4.2: IoF message header. While the TBR supports a full 2 bytes for their serial numbers, 14-bits should provide plenty of range to cover possible numbers for a long time to come. The flag is set to 0 if there is no SLIM packet, and it is set to 1 if the first packet of the payload is a SLIM packet. Flag value 2 and 3 are reserved for potential future flags. The reference timestamp is set to be the first detection registered from the TBR, but if there has been no detections, and SLIM packet is set to send, the reference timestamp is set to the GPS timestamp.

namely the codetype. This is a single byte that contains both the frequency transmitted on, and the communication protocol the tag has used to do so. The encoding of this byte is proprietary and belongs to Thelma BioTel. The communication protocols are however standard. Currently, none of the supported protocols use more than 3 bytes, but in the future there will most likely be introduced new protocols supporting more complex checksums. Table 4.1 shows how the different protocols use bytes to send tag ID and optionally tag data.

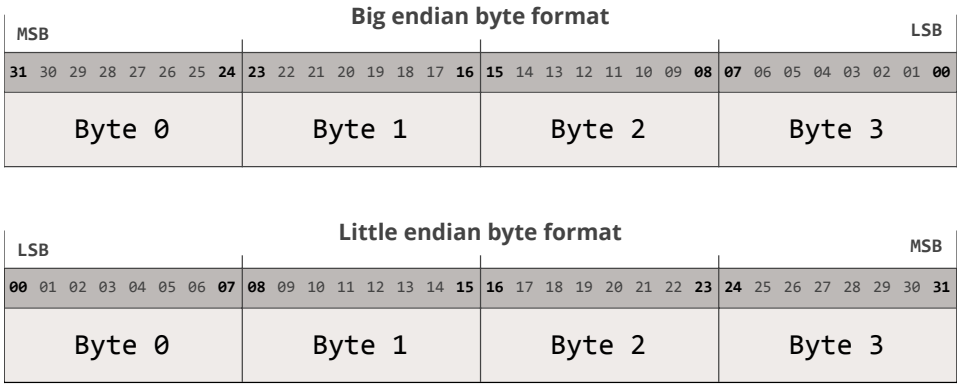


Figure 4.3: Big byte order is user for the IoF messages.

Signal-to-noise ratio (SNR) is calculated using eq. (4.2), where the values will range from 6 to 60, depending on signal strength and noise conditions. 6 means very weak signals, and 30 and above is very strong. The scale is logarithmic, and can be covered by using 6 bits. Lastly, the millisecond precision of time of arrival is included as 10 bits, ranging from 0 to 999. Figure 4.4 shows the bit and byte usage of the tag packet, which typically is 6 bytes large when using the S256 protocol (most common).

Table 4.1: Byte usage of different transmission protocols. In the future, new protocols using more complex checksums, or larger ID/data, might be added.

Combinations of tag ID and tag Data

Communication Protocol	Byte 00	Byte 01	Byte 02
R256	Tag ID	empty	
R04K	Tag ID (12-bit)	empty	
R64K	Tag ID		empty
S256	Tag ID	Tag Data	empty
R01M	Tag ID (20-bit)		empty
S64K	Tag ID		Tag Data
HS256	Tag ID	Tag Data	
DS256	Tag ID	Tag Data 1	Tag Data 2

$$SNR\,[\text{dB}] = 10\log_{10}\left(\frac{Average\ peak\ signal\ power\ in\ pulse\ train}{Average\ noise\ power}\right) \tag{4.1}$$

$$= 20\log_{10}\left(\frac{Average\ peak\ signal\ amplitude\ in\ pulse\ train}{\boldsymbol{Average\ \ noise\ \ amplitude}}\right) \tag{4.2}$$

Tag detection packet

Field name	Seconds passed since reference timestamp	Codetype	Tag ID	Tag Data [Optional]
Allowed values	[0, ..., 255]	[0, ..., 255] *proprietary	[0, ..., 1048575]	[0, ..., 65535] →
Field bit value	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	.....	.....
Byte bit value	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Byte	Byte 00	Byte 01	Byte 02 - Byte n	Byte n+1 - Byte m

SNR	Millisecond
→ [6, ..., 60]	[0, ..., 999]
5 4 3 2 1 0	9 8 7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Byte m+1	Byte m+2

$n = 2, 3, 4$

m: $n \leq m \leq 4$

Figure 4.4: Tag detections packet. If it is the first packet in the payload, byte 00 will be equal to 0. And as can be seen, tag detection packets will at least be 5 bytes, and at the most 7 bytes, depending on the communication protocol.

4.1.4 TBR sensor packet

Like the tag packet, the first byte of the TBR detection packet is a byte that gives the number of seconds passed since the reference timestamp set in header, and it will likewise be 0 if the TBR packet is the first (i.e. reference timestamp set as TBR timestamp). The second byte is also the same as the tag packet, codetype, just that for TBR it is always set to 255, an otherwise unused codetype value, to indicate that it is a TBR message. This is followed by 2 bytes temperature, measured by a factory calibrated temperature chip inside the TBR housing, which is converted in back-end using eq. (4.3). Finally, the TBR packet

consists of 1 byte ambient noise average, 1 byte ambient noise peak, and 1 byte frequency. Figure 4.5 shows the bit and byte usage of the TBR packet, which is 7 bytes long.

$$Temperature\ [°C] = \frac{data - 50}{10} \tag{4.3}$$

TBR sensor packet

Field name	Seconds passed since ref. timestamp	Codetype	Temperature	
Allowed values	[0, ..., 255]	[255]	[0, ..., 65535]	→
Field bit value	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
Byte bit value	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Byte	Byte 00	Byte 01	Byte 02	Byte 03

	Noise avg	Noise peak	Frequency [kHz]
→	[0, ..., 255]	[0, ..., 255]	[63, ..., 77]
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
	Byte 04	Byte 05	Byte 06

Figure 4.5: TBR sensor data packet. If it is the first packet in the payload, byte 00 will be equal to 0. The packet is 7 bytes long, and there are no fields that require bit manipulation.

4.1.5 SLIM packet

The SLIM packet involves the most bit manipulation of the three packet types, in order to reduce the total size. This is partly due to precision of longitude and latitude not fitting naturally within complete bytes. Longitude goes from −180 to 180 degrees, so at least 9 bits are needed to cover the 360 possible values. Of course, one could do a sort of optimization where you limit the longitude to one hemisphere, or even smaller areas, but this requires specialized implementations. To have precise positioning, the GPS should be accurate to around 1 meter. 4 decimal places gives around 11 meters accuracy, while 5 gives accuracy up to around 1.1 meters [38]. To have 360 degrees with 5 digits precision, 26 bits is needed. Similarly, for latitude, which goes from −90 to 90, 25 bits is needed (8

for degree, 17 for precision). 14 bits have been set aside for SLIM status messages, like battery status, but currently it only sends a static number until this is implemented in SLIM software. This was not a priority during the project development. When getting a GPS fix, other parameters are also set, like a measure of position dilution of precision (PDOP). A lower value is good, and after a certain point most GPS messages can be discarded if they have higher PDOP values (above 10 is only for rough estimates of location, above 20 is poor). Setting the threshold to 12.7, we can represent the value with 1 decimal precision using 7 bits. Ideal is in that case 1, while 12.7 is worst-case scenario. We also get a measure on the fix quality, where the ideal possible values can be seen in table 4.2. 5 numbers can be represented with 3 bits. Lastly we have a measure of the number of satellites tracked for the position fix. 5 bits will cover up to 32, which cover all possible Navstar satellites. Figure 4.6 shows the bit and byte usage of the SLIM packet, which is 10 bytes long.

Table 4.2: Possible GPS fix values. The table of values is from [40].

0	No fix
1	Dead reckoning only
2	2D-fix
3	3D-fix
4	GNSS + dead reckoning combined
5	Time only fix

4.1.6 New IoF protocol vs previous version

In the previous version, the header was 2 bytes (no reference timestamp), SLIM packets were not implemented, and tag and TBR packets were always 11 bytes long [19]. This made unpacking easier, as you could separate each IoF message in equal 11 bytes long parts. The packets were 11 bytes long because timestamp was included as a 4 byte value in each packet, and for tag packets, tag ID and Data were allocated as 4 bytes every time, 2 each, covering all possible communication protocols (except R01M). In addition, TBR had an empty byte to make it 11 bytes long. This meant that 10 S256 tag detections as payload with the old protocol would use 112 bytes: $2 \cdot 1 + 10 \cdot 11 = 112$ [bytes].

In the new protocol, the header is always 6 bytes, the SLIM packet is always 10 bytes, and TBR is always 7 bytes. Tag detection packets size depend on communication protocol

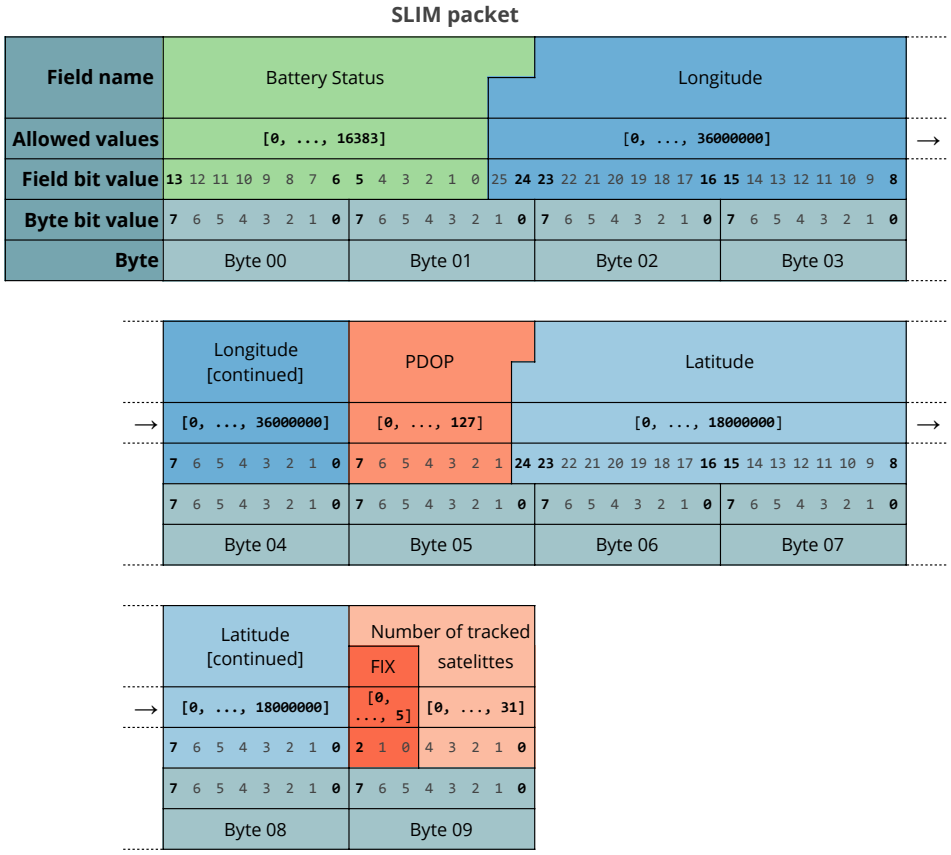


Figure 4.6: SLIM packet.

used. Currently, the most commonly used protocol is S256, meaning that the tag packets will be 6 bytes. Given that the SLIM is set to send a new message each minute, and that an S256 message takes approximately 4 seconds, the maximum amount of tag detections for a minute is 15. This will of course be in very favourable conditions, for the TBR to successfully detect each packet straight after the other without disturbance from other tags. Add in a SLIM and a TBR sensor packet, and we get the approximate maximum size of an IoF message: $6 \cdot 1 + 10 \cdot 1 + 7 \cdot 1 + 6 \cdot 15 = 113$ [bytes]. So, 5 more tag detections, plus TBR sensor and SLIM packets, while using only 1 more byte than the previous protocol. This is a great improvement. A more involved communication protocol like HS256 would take longer to transmit per message, and so you would not achieve 15 messages within a

minute, but for argument's sake we could allow it. Then the payload would be 122 bytes, making the total message size 128 bytes, which is exactly the data limitation per message for the chosen LoRa setup. 15 HS256 detections and a TBR sensor packet in the old format would use 178 bytes, still without SLIM packet.

A benefit of the old protocol is the ease of unpacking equal-sized packets. The greatest advantage with this approach being that you can keep unpacking the rest of an IoF message if one of the packets are corrupt. Unfortunately, you cannot do that with the new protocol, as the sizes vary depending on the communication protocol used. Say a packet is corrupted, and the codetype is set to another communication protocol than the actual protocol used, i.e. R256 instead of S256. That would mean that the parser unpacking the message would think there is no tag data, only 1 byte tag ID, and would unpack the rest of the message accordingly, effectively shifting all the other values 1 byte off. If really unlucky, the corrupted packet is first in the payload, making the whole payload unreadable.

However, corruption of packets does not happen often, so the amount of lost messages are acceptable. This was verified for a separate experiment based on the same case study as the one of this thesis, where packet error rate (PER) of the IoF concept was defined as the ratio of packets lost in transmission from nodes (SLIM) to server (MQTT server). PER is based on two types of losses, one due to radio interface (perception to network layer), and one due to internet loss (network layer to application layer). All SLIM nodes achieved a PER of less than 8% [17]. The ratio is made by comparing the amount of messages transmitted to the amount of messages that was received successfully in the application layer. The number of actual corrupted messages is lower.

4.2 Persistent storage

There exists a vast amount of solutions for persistent storage, like storing bytes directly to `txt`-files, relational databases (typically SQL), non-relational databases (NoSQL databases), in-memory databases, and navigational databases to name a few. These are all different principles behind how to store data persistently, and for each there exists many solutions open-source and commercial, spread across a number of programming languages. Focusing on databases, a normal way to describe expected properties of database transac-

tions is ACID (Atomicity, Consistency, Isolation, Durability) [15]. If you stop persisting a database, Durability is lost. Atomicity is used for committing changes to a database as an all or nothing operation. Isolation is describing what part of transnational changes is seen by other processes, which in this context means that processes should never see partial transactions. The last attribute, Consistency, is that the description of the structure and relationship of the data in the database should be maintained for all writers and readers of the database.

Relational and non-relational are probably the two most widespread. The main difference worth noting between them is that non-relational databases do not require a predefined schema to store data, and thus allows dynamic changes to a database structure. Rather than scaling up, i.e. increasing server size to tackle more data load, non-relational scales out: the database can be distributed across multiple servers as load increased. This makes non-relational databases great for scaling, consequently becoming increasingly popular in big data, which is also typically more dependent on dynamic changes to data structuring. Non-relational databases are however often not compatible with ACID. Relational databases are mostly all compliant with the ACID principle. This alone can be reason to choose a relational database solution, network distributed if necessary.

Another solution, in-memory databases, is typically not ACID-compatible either, since they are not persistent. So durability of ACID is lost, making it susceptible to data loss in case of faulty operation. Most implementations have however some sort of persistent storage backup or underlying database. RAM is more expensive than disk storage, as most computers have much less RAM than disk space, but RAM is on the other hand much faster than disk. For smaller amounts of data, which is read and handled often, in-memory database solutions might be a great way to handle the data. This is given that enough RAM is available, and that backup data in disk storage or are willing to risk loosing data. When it comes to loss of persistence, in-memory databases can run at full speed and maintain data with the introduction of non-volatile random access memory technology [26].

Data amount in an IoF concept implementation will most likely not be of immense size, considering the amount of tags possible in a sea cage or wild setting is limited. The operational life of the tags are also limited. If a commercial sea cage would like to keep

hydrophones and SLIMs in place, only replacing batteries and tagging new fish, then the data would grow over time, although it would also be reasonable to have separate storage for the separate instrumentation periods. Something else to keep in mind is that as of now, there is no apparent reason for allowing end-users to write to the database. The back-end is the only permitted writer to the database, anyone else would be readers. Lastly, the application will run in one location, and be accessed by potentially multiple end-users, but the amount of end-users will in most cases be quite limited. These are some important factors to consider when choosing an appropriate storage solution.

So, while there are clear advantages with a distributed non-relational database when it comes to web applications, relational databases will cover the IoF use case without trouble. Considering that the data load is relatively small, even simple SQL databases can be utilized, i.e. databases that do not support concurrent writers or distributed storage. Another interesting and relatively modern solution is as mentioned the in-memory database. While there is no need for critical response-times, the end-users will want to perform analysis on a relatively large time frame of data, an expensive process to do multiple times if reading from disk each time. Utilizing multi-core processor which can address RAM concurrently, the performance gains can be significant. On the other hand, end-users will probably not access the data constantly, making the cost of keeping data in-memory less valuable than reading from database, if RAM usage is a limiting factor.

4.2.1 Database table structure

Tables 4.4 to 4.6 shows the database table construction chosen for each data message. For the IoF implementation, SQLite was chosen, which is a relational SQL database. SQLite supports 5 data types: `NULL`, `INTEGER`, `REAL`, `TEXT`, and `BLOB` [37]. For the tag table, all columns is of the `INTEGER` type, except Tag Data, which is `REAL`. Similarly, for the TBR table, all columns are `INTEGER` except Temperature, which is also `REAL`. In the SLIM table, Slim Status are `TEXT`, and since the integer value of GPS fix is converted in back-end, so is the Fix column. Latitude, longitude, and PDOP are all `REAL`. The remaining columns are `INTEGER`. Lastly, for the positions table, the timestamp, Tag ID, Frequency, and Millisecond is `INTEGER`. X, Y, Z, Latitude, and Longitude are `REAL`. The last column,

Cage Name, is naturally of type `TEXT`.

Table 4.3: Tag detection database table.

Tag detections Database Table

Message ID	Timestamp	Tbr Serial Number	Comm Protocol	Frequency	Tag ID	Tag Data	Tag Data Raw	SNR	Millisecond
5	1538823331	110	S256	69	35	7.6	38	10	330
5	1538823347	110	S256	71	100	5.2	26	25	450
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
5000	1539523337	112	R256	69	34	Null	Null	32	713

Table 4.4: TBR sensor database table.

TBR sensor Database Table

Message ID	Timestamp	Tbr Serial Number	Temperature	Tempreature Raw Data	Amb. Noise Average	Amb. Noise Peak	Frequency
25	1538823600	112	7.8	128	15	45	69
52	1538824200	112	7.7	127	15	39	69
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
5031	1539523800	110	7.3	123	52	87	69

Furthermore, SQL supports constraints, like the `PRIMARY KEY` constraint, which requires that each record of a table can be uniquely defined. They cannot be `NULL` (they can in SQLite), and each table can only have one primary key, consisting of a single column or multiple columns. `PRIMARY KEY` requires that unique values are contained within the column(s). There are no primary keys defined for any of the IoF tables, as there are no columns that are uniquely defined. However, this is a flaw in the database implementation design. While developing, it was believed that `PRIMARY KEY` is only possible for a single column. It was close to the end of the thesis period that it was discovered that multiple columns can be used. For single columns it does not work, since multiple tag detections share the same message ID, and timestamp too can be shared. However, including TBR

Table 4.5: SLIM database table.

SLIM Database Table

Message ID	Timestamp	Tbr Serial Number	Slim Status	Latitude	Longitude	PDOP	Fix	# Sat. Tracked
25	1538823600	112	75%	63.41850	10.39946	1.4	3D-Fix	12
52	1538824200	112	74%	63.41849	10.39945	2.3	3D-Fix	8
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
5031	1539523800	110	60%	63.41846	10.39941	13	2D-Fix	3

Table 4.6: Positions database table.

Positions Database Table

Timestamp	Tag ID	Frequency	Cage Name	Millisecond	X	Y	Z	Latitude	Longitude
1538823331	35	69	IOF Cage A	325	12	13	12	63.41852	10.39949
1538823347	100	73	IOF Cage B	100	21	8	4	63.41849	10.39953
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1539523337	34	69	IOF Cage A	700	7	30	18	63.41851	10.39951

information would and for example tag ID would make it possible. The advantage of using primary keys are quicker search results, and more powerful cross-table filtering, since it can be used together with `FOREIGN KEYS`. Foreign keys is a column or a collection of columns in a table that refers to the primary key in another table. So one could use the message id as a primary/foreign key pair for example. Message ID is determined by back-end, and is a single upwards counting number for each full IoF message that is received. SQLite also has an internal `RowID` column by default, which automatically increments for each row in a table.

Another SQL constraint is `NOT NULL`, which as the name implies rejects database transactions if there is no data inserted for the column(s) with the constraint. All the

columns in all the IoF tables use this constraint, except the latitude and longitude columns, tag data, and the converted temperature column. This is in case no latitude and longitude is provided in error cases, and in case of missing metadata conversion factor for tag data, or project implementations using tag ID only tags. Finally, the `UNIQUE` constraint is used in the positions table to require that any database insertion has a unique combination of tag ID, timestamp and frequency. This is in case of multiple attempts at positioning the same set of data.

4.3 Components of the system

In this section, the materials and methods used for the implementation of the IoF system is described in more detail.

4.3.1 Thelma BioTel tags and hydrophones

For this system, Thelma BioTel acoustic tags and hydrophones are utilized (fig. 4.7). The tags can transmit data such as: ID, pressure/depth, temperature, acceleration, tilt, dissolved oxygen, conductivity, and salinity. The acoustic receiver, TBR-700 Real-time acoustic receiver, is designed to operate at 69 kHz. However, it supports full multi-frequency reception in the 60-80 kHz band [6].

4.3.2 Synchronization and LoRa Interface Module (SLIM)

A custom surface module is used for this system. The module includes GPS for precise positioning and time synchronization, a LoRaWAN interface for LoRa communication, and a RS-485 interface for communication with a TBR 700. In addition, the module has a small low-powered screen for easy information gathering while running / debugging. These are put together on a PCB board and in a water-proofed box (fig. 4.8). The GPS is important for positioning of the acoustic tags, as it will allow the different hydrophones to synchronize their time accurately, counteracting clock drift, and it gives known positions needed for the positioning principle described in section 2.2. The module currently supports the new IoF protocol format described in section 4.1. The module is still in development.

(a) **(b)**

(c)

Figure 4.7: Thelma BioTel acoustic tags and hydrophone. Images from [6]. **(a)** Acoustic tags of varying sizes. **(b)** A 9 mm acoustic tag of type LP-9. **(c)** A TBR 700 RT

4.3.3 Gateway

The surface module provides a connection to the hydrophones in the water from the on-site location. The data received there needs to be forwarded until it reaches the MQTT server. As mentioned before, the surface module has a LoRaWAN interface, which it uses to send LoRa messages to the gateway on-site. The gateway should preferably be placed somewhere stationary with a constant internet connection. Due to power outages and poor internet connections at site locations, this could preferably be done using a battery and mobile network card, rather than power and an Ethernet cable. Naturally, the gateway should be placed within listening range of the surface modules. The gateway needs to be capable of receiving LoRa messages, unpacking them and sending them onwards as an MQTT message to the server. The gateway will work as a MQTT publisher in the network. For this implementation, the MultiConnect© Conduit™ from Multitech Systems is used (fig. 4.9) [3].

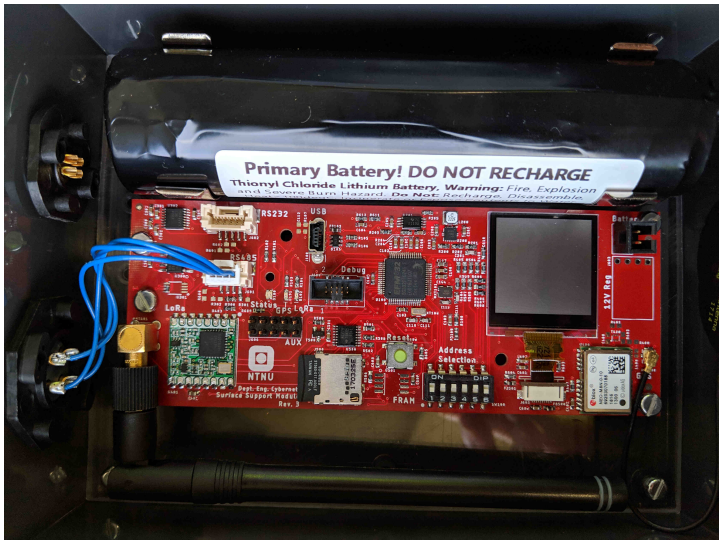


Figure 4.8: Image of surface module designed by PhD candidate Waseem Hassan and associate professor Jo Arve Alfredsen. The module is still in development. Photo by Per Arne Kjelsvik, the author of this article.

Figure 4.9: MultiConnect© Conduit™

4.3.4 MQTT Server

A MQTT server is needed for receiving the data in the end of the system. A `Mosquitto` MQTT broker for Python was set up on a `18.04 Ubuntu LTS` virtual server at NTNU. Any MQTT client of choice can then connect and subscribe to the server. `Mosquitto` was chosen because it is developed and run in Python, and is a stable and established version of a MQTT broker.

4.4 Software tools and libraries

For the system implementation, a Python MQTT client was written using the `paho-mqtt` library. Here, the client unpacks the data and stores it in a readable manner in a SQLite database. For the front-end of the system, the Python library `dash`, developed by Plotly. was used to create a web interface. The end-product is a website the end-user can use to plot data interactively. Figure 4.10 shows all the libraries chosen for IoF. Below is a description on the reasoning behind choosing the different libraries, and what they do.

4.4.1 Back-end libraries

As mentioned in chapter 3, it was desired to keep the code footprint minimal, relying on a minimal amount of libraries and tools, preferably open-source. Another desired requirement was to avoid relying on program features / tools that would quickly become obsolete, but also avoid libraries early in development where support is limited and updates might break functionality often. Python released the first stable release of version `3.7` summer 2018. While fairly new, this is a good place to base the software since it will mean small amounts of maintenance for a long time to come. Upgrading to new releases of Python will in turn also be easier.

The chosen MQTT client and broker are native to Python, and are both developed open-source under the Eclipse foundation. While other MQTT client libraries exist, of the open-source ones, `paho-mqtt` is one of the most popular with a long stable history and development team behind it. The major version of MQTT currently widely in use is version 3.1.1. The next major version, version 5 (version 4 was skipped), was released spring 2019, becoming an official OASIS standard [29]. This version brought many needed changes, like better error reporting, message metadata, load balancing, and most importantly improved security. This is a fresh release, and it is not fully supported in all MQTT brokers and clients. For example, only the `C` version of `paho-mqtt` supports version 5 at time of writing. Version 3.1.1 was therefore used, which is true and tested to work, but upgrading to version 5 should be done when it is possible to do so.

SQLite is a simple database that is built directly into Python (`sqlite3`). Considering the needs of the IoF concept discussed in section 4.2, SQLite should be sufficient. While

simple, it is capable of handling large amounts of data and a fairly large amount of traffic when used for a web application. Since the database resides in one file, it is easy to transfer, should the need to view the data offline for end-users come up. If there is a need for many concurrent writes, big data, or if the web application is located in another network, then a client/server database (for example PostgreSQL) would be more suited. Otherwise SQLite is a fast and reliable solution.

TOML is short for *Tom's Obvious, Minimal Language*. It aims to be a minimal configuration file format that is easy to read, designed to map unambiguously to a hash table. It is supported by many programming languages, including Python. For IoF, `toml` was chosen for metadata and configuration files. MQTT broker settings, database location and names, and metadata such as tag IDs used in a project is stored in `toml`-files.

(a) Virtual linux server

(b) Python 3.7.1

(c) mosquitto broker

(d) paho MQTT client

(e) SQLite

(f) toml

(g) Flask

(h) plotly

(i) plotly Dash

(j) NumPy

(k) pandas

(l) Apache arrow

Figure 4.10: Tools and libraries used for the software development of this system, all contained to the Python programming language (except the virtual server which is a Linux operating system). Several libraries and tools are shared between both front-end and back-end. **(g)** - **(i)** and **l** are unique to front-end, and **(c)**, **(d)** and **(f)** are unique to back-end.

NumPy is a well-established Python library for scientific computing in Python. It provides support for multidimensional array objects, and several fast operations on arrays. For IoF, Numpy was used in the back-end for positioning, but mostly simple functions were used. However, more importantly, `pandas` is based on `NumPy`. Pandas stands for *Python Data Analysis Library*, and it provides easy-to-use data structures and analysis tools with high performance. It is arguably one of the most used data analysis tools used with Python, especially when it comes to transforming and mapping data from one format to another. Typical workflow for Pandas is to take data, like a CSV file or SQL database, and create a Python object with rows and columns which make up a so called DataFrame. Dataframes are similar in structure to spreadsheet tables, making it easier to work with than lists or dictionaries that needs for-loops to go through. Vectorization is therefore key when it comes to Pandas, meaning to execute operations on entire arrays simultaneously. For the chosen web framework, Dash, it is common to use dataframes such as the one provided by Pandas. Pandas is well suited for the purpose of filtering, searching, and transforming the IoF data. It was also used in positioning, more details on that in section 4.5.

4.4.2 Front-end libraries

Finally, for interactive plotting, Python has multiple open-source solutions and visualization libraries (fig. 4.11). `Bokeh` is a popular library developed by Anaconda. It offers rich and interactive graphs, and has been available for use since 2013. It is using `D3` (JavaScript visualization library) for its front-end, and `Tornado` (Python web framework) for back-end. Dash by Plotly was launched in 2017, uses React (JavaScript library for user interfaces) and Plotly (a visualization library also based on `D3`) for front-end, and Flask (Python web framework) for back-end. Dash is fairly new, and thus it does not support the same breadth of features as Bokeh. However, it is being developed by Plotly, and uses the Plotly library in its core, which is a powerful library that has been developed since 2012 with wide community support. The development speed of Dash is rapid, adding new features often. This increases risk of updates breaking functionality, but it seems more promising long-term than Bokeh. Shortly before finishing this paper, `dash v1.0.0` was released [33]. This release of Dash formalizes a commitment from the dash developer-team to not

introduce any breaking changes until the next major release (`2.0`).

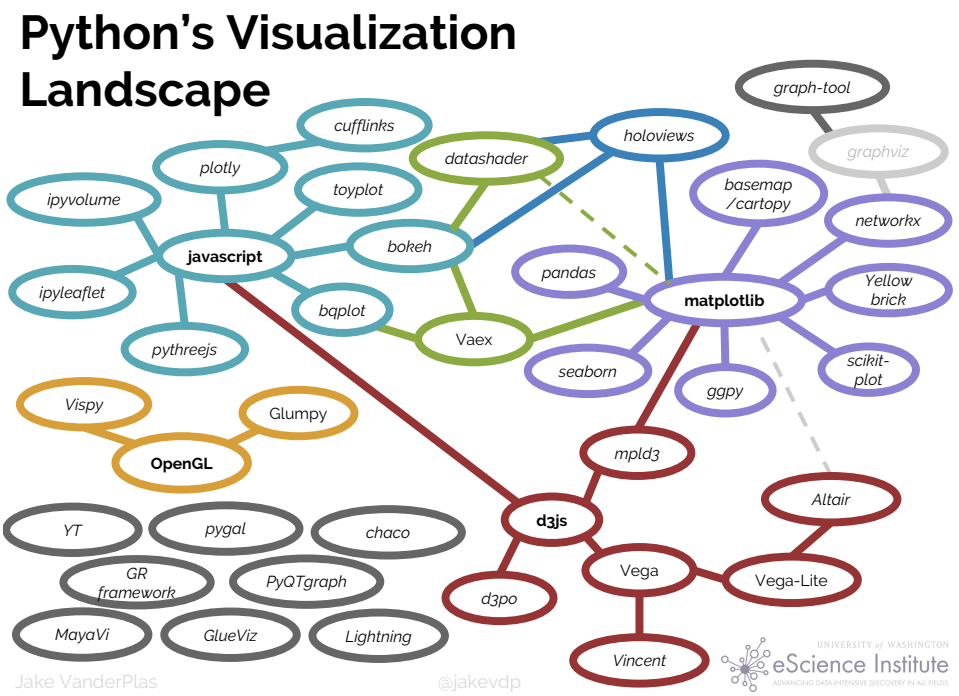


Figure 4.11: Python's Visualization Landscape. Figure shows some of the available options for visualizing data in Python. Not all of these are interactive and web-supported, but one could build interactivity on top. This figure is from Jake VanderPlas of University of Washington, from his speak at PyCon 2017 [30]

For Dash, everything is contained within Python. Dash allows you to use custom data structures such as `pandas DataFrames`, and you have full control of CSS. Adding or writing custom React components can easily be done too. With Bokeh on the other hand, it is designed to use their own data structure `ColumnDataSource` (common structures like `pandas DataFrames` and `NumPy Arrays` are supported too), and some interaction functionality must be written in JavaScript. Dash is based on Flask, a widely used web framework, and since Dash is in principle only a Flask app with some modification, one can easily customize and control all parts of the app. Bokeh is based on Tornado, which is not as widely adopted as Flask, meaning less documentation and less features and customization options. In the end both libraries offers tools that will meet the requirements of IoF well,

but Dash was chosen for these reasons, in addition to being very easy to approach for writing applications neatly and effectively. There exists many more interactive libraries, `pygal`, `HoloViews`, `Chartify`, `Tableau`, `Microsoft PowerBI`, and `Altair` to name a few. And new ones are announced and released often too. It is a rapidly developing landscape of options, but Bokeh and Dash stands out as the two most viable open-source options, with the greatest support for web interactivity and chart types in Python.

4.5 Code structure and implementation

In this section, details on how the code for back-end and front-end is structured and written will be given. However, low-level details on specific implementation is partially omitted. A general structure of the IoF software is presented briefly, as well as logical flow of the system. The project code has been written with Python Enhancement Proposal (PEPs) in mind, such as PEP8 that describe a style guide for Python code [42]. Using the relatively new Python type hints has also been attempted, but it has not been prioritized to have the code work with static type checkers like `pytype`, although this would be beneficial to do in the future. The back-end code is generally more well documented, as more development time of front-end was spent on showcasing the specific case study. The full project code is available in a GitHub repository, see [20].

4.5.1 Back-end

In the previous implementation of back-end, the unpacking of IoF messages relied on fixed-size packets. It proved to be difficult to extend it to work with the new format, and so it was better to completely redo the message handling of IoF messages in back-end. The general flow of the program remains the same however. The back-end is constructed as a Python package, with three main files to run the back-end, and two sub-packages. The three files are `main`, `initbackend` and `mqttclient`. `main` initializes back-end (creating databases, configuring MQTT) with `initbackend` through a command-line interface and arguments parsed when calling `main` from the terminal. Any configuration files used in the back-end are coded with the TOML format [34]. After initialization is complete, `main`

spawns `mqttclient` as a subprocess. Should `mqttclient` crash for some reason, `main` spawns a new client, similar to a process pair solution.

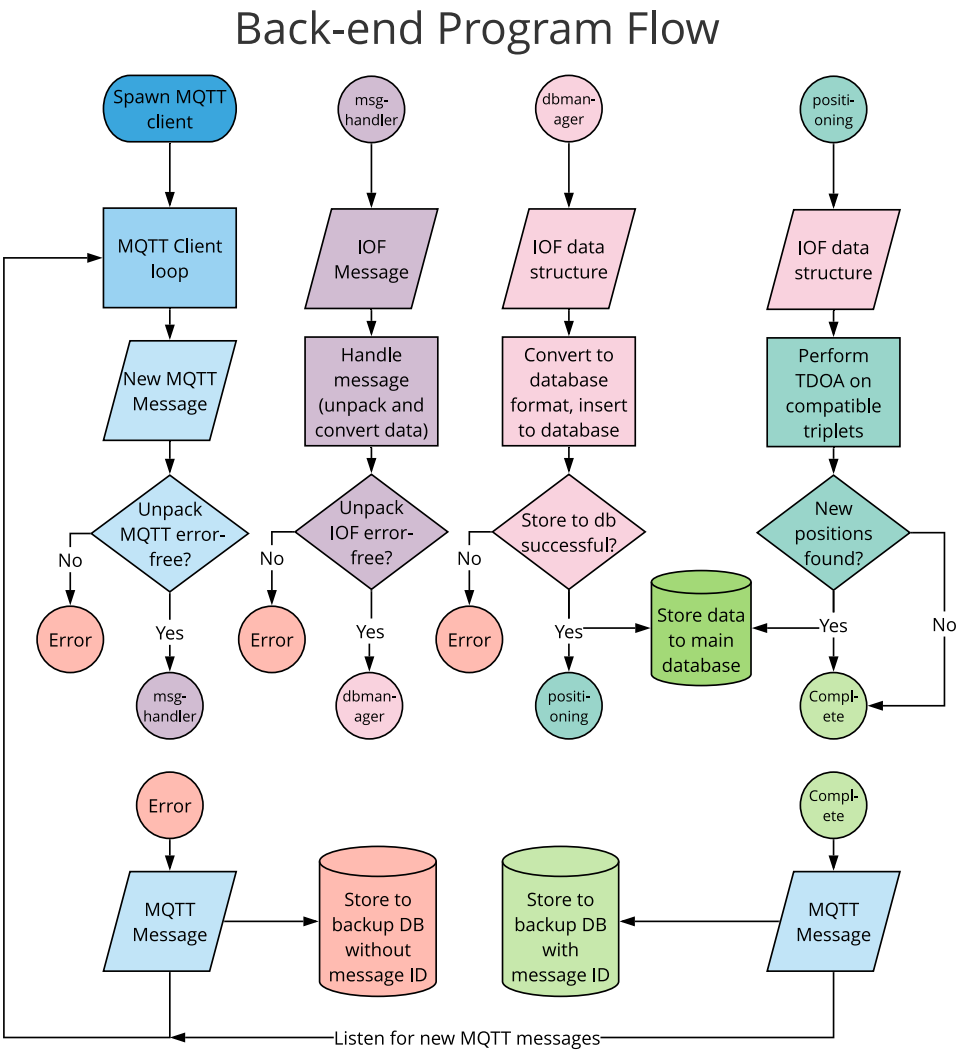


Figure 4.12: Figure showing control logic in back-end software. The front-end would then poll the storage database for the web application.

`mqttclient` is the main process responsible for the back-end, as it is responsible for accepting new MQTT messages, unpacking the raw data, transforming it, and storing it to persistent storage. The client initializes by connecting to the MQTT server and subscribing

to the # topic (subscribing to all topics of the MQTT server). Figure 4.12 shows a simplified view of the logic flow of the `mqttclient`. The message handling is as mentioned separated into a sub-package (*msgHandler*), and so is database management (*dbManager*). Positioning Python modules are coupled together inside the database management sub-package.

Positioning

Positioning is an important component of the IoF system, and quite some time was spent on developing the code for this particular part. The TDOA algorithm is based on [10]. Algorithm 3 shows a pseudocode representation of how the algorithm is implemented. In Appendix B the full implementation is shown with well documented code. Note that there is no need for a separate case for $R_{ac} = 0$ when $R_{ab} \neq 0$, like explained in section 2.2.

TDOA-HYPERBOLA-POSITIONING(*timestamps*, *stationData*, *depth*)

```
1  // Initialize algorithm variables from input
2  positions = None
3  if R_ab == 0
4      x = b/2
5      x-candidates = [x, x]
6      if R_ac == 0
7          y = (c^2 − 2c_x x)/2c_y
8          y-candidates = [y, y]
9          positions = Array([x, y, depth])          // positions is a 1D array
10     else
11         // Calculate factors a_2, a_1, a_0 and find roots of y-polynomial
12         y-candidates = roots([a_2, a_1, a_0])
13 else
14     // Calculate variables g, h, d, e, and f
15     x-candidates = roots([d, e, f − z^2])
16     y-candidates = g · x-candidates + h
17 if positions == None
18     positions = Array(x-candidates, y-candidates)      // positions is a 2D array
19 // If positions is defined and not complex ⇒ valid candidates have been found
```

Algorithm 3: TDOA based positioning. See Appendix B for the full Python implementation with documentation.

Adjusting drifted timestamps of tag detections

When a new message has been stored to persistent storage, positioning can be performed. Back-end uses the message, reads the last n numbers of messages, and looks for a matching pair of each tag ID in the current message. It then performs TDOA-positioning using algorithm 3. Another use-case is to perform positioning for a complete database. This is a bit more involved because it has to to find all triplets for a given tag ID within relatively large amounts of data. Should one try to loop through all data for each tag ID it would be very resource-demanding and slow.

Here Pandas is a good fit, to use vectorized operations to search the database data quickly. However, one issue is that the GPS synchronization can have a timestamp drift of 1 or 2 seconds. This is a flaw in SLIM that should not occur. The issue has not been fully resolved at time of writing. In most cases it will be perfectly synchronized, but in some cases one of the timestamps will be off by a second, and in a few worst cases, all three are off, so the largest is 2 seconds ahead of the smallest.

Hence it became apparent that a method to identify normal triplets, as well as drifted triplets (that we adjust to the middle timestamp) was needed. One needs to be careful to not adjust valid "drifts", meaning cases where the detection occurred right around the timestamp rolling over to a new second. This can be avoided by assuming that tag detections cannot reach farther than a given distance, for example 1 km. 1 km with 1500 m/s sound speed underwater maps to 667 milliseconds. Figure 4.13 illustrates the DataFrame transformation we want to achieve for these cases.

During the thesis development a procedure was implemented that finds tag triplets. The full code implementation with documentation of this procedure is included in Appendix C. Bear in mind, the core challenge is to find triplets fast and efficiently. To adjust the timestamps at the same was an added challenge. The procedure accepts a `pandas DataFrame` as argument, which it operates on to extract triplets. In the case of a new message, the input data will be a single triplet. If it is a valid triplet, no adjustment is made. In the database case, the input data will be a large object with all detections of a given tagID as separate rows. It will then proceed to find triplets.

First the dataframe is sorted by timestamp in ascending order, and the row index is

Timestamp drift fix illustration

	Timestamp	Millisecond	TBR Serial ID	Tag Data
0	1556555369	325	45	3.5
1	1556555370	330	47	3.5
2	1556555371	335	49	3.5

⇓

	Timestamp	Millisecond	TBR Serial ID	Tag Data
0	1556555370	325	45	3.5
1	1556555370	330	47	3.5
2	1556555370	335	49	3.5

Figure 4.13: Timestamp drift adjustment. Here the data is structure like a `pandas` DataFrame, meaning a row index on the left, and column index at the top. The tables illustrate the fix of a worst case drift scenario of a tag detection triplet. Tag Data means depth in this case.

reset. The timestamps are then extracted as a series, and a function is used on the series to find all rows where the difference between that row and two rows ahead is less than a threshold (2 seconds). The indices of the highest row is stored in a list. Another list is made with the same indices, and the indices of the other rows as well. A mask column is then added to the dataframe, where the rows with the triplet indices are set to an increasing number. So, in the mask column, triplet number one will have mask value 0, triplet number two will have 1, and so on. All rows with no value in the mask column are removed from the dataframe, keeping only the triplets. We are now ready to adjust the timestamps.

Next, another column *drift* is added to the dataframe, which consists of $timestamp + millisecond/1000$ for each row, and the difference between rows in drift is extracted as a new series. Each row keep the index in the original frame, and each row contain the drift difference between that index and the next one. By comparing the drift differences to the millisecond threshold (0.667), we can extract all indices where the third row of a triplet is drifted 1 second after the middle row, and all indices where the first row of a triplet is

drifted 1 second before the middle row. These indices are stored in separate lists, and then used to adjust all matching timestamp columns in the dataframe to be ± 1 second (set equal to the middle row timestamp of a triplet). Figure 4.13 shows how the dataframe looks in this stage of the procedure (invalid triplet columns are kept in the illustration for clarity, though they would be removed before adding drift).

Triplet mask and drift dataframe

	Timestamp	Millisecond	TBR Serial ID	Tag Data	Mask	Drift
0	1556555369	325	45	3.5	0	NaN
1	1556555370	330	47	3.5	0	1.005
2	1556555371	335	49	3.5	0	1.005
3	1556555440	405	47	7.5	NaN	69.070
4	1556555440	415	49	7.5	NaN	0.010
5	1556555519	473	49	17.2	1	79.058
6	1556555519	483	45	17.2	1	0.010
7	1556555520	485	47	17.2	1	1.002

Figure 4.14: Triplet mask and drift dataframe. The first row of a triplet is never checked, since it will always be large. Row index 0-2 the first triplet in the dataframe and so mask is set to 0. As can be seen in the drift column, both row index 1 and 2 are 1.005 in difference, meaning both are greater than 667 milliseconds. This means that timestamp of row 0 is set +1, and timestamp of row 2 is set −1. Row index 3 and 4 is lacking a third detection, and so the mask is not set, since it is not a valid triplet. These invalid rows are removed after masking in the actual implementation, and thus the drift would never be calculated for them, improving performance for large dataframes. Row index 5-7 is the next triplet (mask value 1), but here it is only row index 7 with a timestamp with more than 667 milliseconds drift. So, timestamp of row 7 is set −1.

Finally, a list is made with each triplet as a separate `pandas DataFrame`. The extraction is achieved by grouping the rows by the mask value, extracting each unique mask value as a separate from, dropping the mask and drift columns at the same time. By placing the triplets in separate dataframes, we have the same structure of the triplets as the case for each message that comes in, i.e. a dataframe for each single triplet. Positioning is then a

matter of looping through the list of triplets and performing TDOA on each one.

4.5.2 Front-end

This section is used to explain the general principle of how a Dash application works. And here a brief discussion on some of the challenges with Dash faced during development is explained in more detail too. Two shorter subsections are presented on the design and construction of the IoF application. The main result of the front-end web application can be seen in fig. 4.17, and [21] provides a short Youtube playlist demonstrating the app live.

The MVC model and Dash

Dash and the underlying Flask library, is like many other web frameworks based on the MVC model (Model-view-controller), first generalised as a concept in [22]. MVC is a principle where the application is divided into interconnected parts, increasing modularity since internal handling of data and functions is hidden within each part. The controller takes input and uses it to send commands to the model or view. The model stores and manages the data. It is the core of the application. The view is what the users see, so for example the visualization of data in the form of graphs. Figure 4.15 shows a simple illustration of the MVC principle.

While developing the back-end code, an attempt was made to decouple the different packages so that it would be easy to change internal handling without breaking the other parts of the back-end. This approach resembles in some ways the MVC model, but it is not directly translatable, as there is no clear definition of the view, and controller is spread across the model and view.

Front-end is on the other hand following the MVC pattern. A typical Dash application is structured in three parts. First is initialization of data sources, parameters, and data manipulation functions. Second is the app layout, which needs to be predefined for all Dash apps. The layout supports full html and css customization through `dash-html`. Third is the Dash callbacks. Dash is built with callbacks in mind, so that every time a user interacts with the web application, a callback function is triggered to execute desired changes and feedback the result. This is what gives the interactivity Dash thrives on.

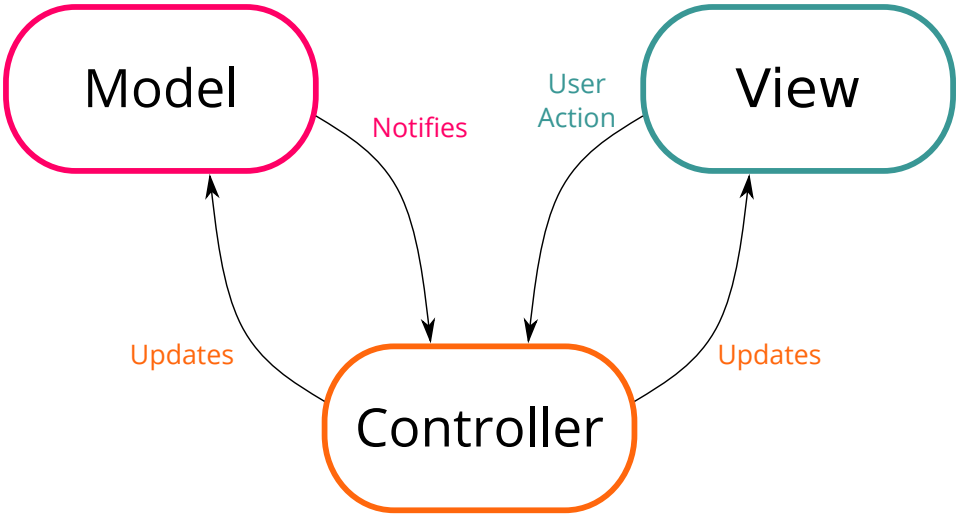


Figure 4.15: Model-View-Controller (MVC) principle. User sends input to controller through view, controller commands model to update data. Model updates controller of new data, controller updates view with new data.

With this it should be clear that Dash is directly translatable to the MVC pattern. The view is represented by the Dash layout, hosting all the html code that the end-users see like graphs and menu buttons. The view sends user inputs as input to the callback functions. They act as the controller of MVC, transforming the inputs to commands for data manipulation. The functions that deal with filtering and other data manipulation is then representing the model of MVC. They return the results to the callback functions. The callback functions proceed to convert the results to html code and returns it to the view again.

Dash has two core principles. The first is that UIs are assembled with declarative components. The components are completely described by their properties. The second principle is that UIs are updated with reactive, functional callbacks. This gives a simple interface for observing and updating any property of any component. Since the callback functions are normal programming language functions, they can do mostly anything. The declarative UI components are defined in the app layout code (view), and the same components are updated with the callbacks (controller) calling or executing underlying data processing methods (model).

The callbacks functions of Dash accept three inputs: `Output`, `Input`, and `State`. One can have multiple of all of these in a single callback, but there can only be one `Output` in total for each html layout component. The `Output` is what the callback function returns to, specifically which property or which component. The `Input` are components in the webpage that will immediately trigger the callback function to execute. `State` is the current setting of a component. An example use-case of `State` is to include the current value of a dropdown value as `State` to a callback, so that it is used for processing there. Had it been included as `Input`, the callback would be triggered immediately after the user selected the dropdown option. Simplified generalized Python code illustrating the general Dash app structure is included in listing 1.

Sharing data between callbacks

Global variables will break a Dash app. In the Dash documentation, it is explained well:

> "*Dash is designed to work in multi-user environments where multiple people may view the application at the same time and will have independent sessions. If your app uses modified global variables, then one user's session could set the variable to one value which would affect the next user's session. Dash is also designed to be able to run with multiple Python workers so that callbacks can be executed in parallel. [...] When Dash apps run across multiple workers, their memory is not shared. This means that if you modify a global variable in one callback, that modification will not be applied to the rest of the workers.*"
> [7]

The documentation goes on to explain that in order to share data safely it is needed to store it somewhere accessible for each process, and that there are three main places to store this data:

1. In the user's browser session
2. On the disk (e.g. on a file or on a new database)
3. In a shared memory space like with Redis

It is relatively easy to handle and avoid in small applications, but as data size grows, it can

```
import dash  # Core dash application API
import dash_core_components as dcc   # Core components (dropdown menus etc.)
import dash_html_components as html  # Used to declare app layout
from dash.dependencies import Input, Output, State  # Used in callbacks
import plotly.graph_objs as go  # Plotly visualization library


app = dash.Dash(__name__)

# Part 1: Model
# |-- Data processing methods, parameters initilization etc.
def retrieve_advanced_dropdown_options(optionType, variables)
    # Perform processing of data
    return advancedOptions[optionType][variables]


# Part 2: View
# |-- The app layout (webpage design) is declared in a html.div
app.layout = html.Div([
    dcc.Dropdown(id="options-selection-dropdown", value=basicOptionsList),
    html.Div(id="display-text-box", children="Basic options enbaled"),
    # [...] more code that declares further layout
])


# Part 3: Controller
# |-- The callback functions updating the app layout
@app.callback(
    [
        Output("options-selection-dropdown", "options"),
        Output("display-text-box", "children"),
    ],
    [Input("enable-advanced-options", "value")],
    [State("options-filter", "value")],
)
def update_plot_options(advancedOptions, optionsFilter):
    # [...] code to handle the callback
    return advancedList, "Advanced plot options enabled!"


if __name__ == "__main__":
    app.run_server()  # Starts app locally: http://127.0.0.1:8050
```

Listing 1: The code only illustrates app structure. In this case it does not do anything, and it is non-logical advanced handling of a dropdown, but general app flow is seen in this example.

become a challenge to solve. The challenges involved with this problem were experienced during the development of the use-case Dash app, where the data grew to 10 million rows in an SQLite database from 10th of November 2018 to 10th of may 2019. The Dash documentation also provided some examples on how to solve the issue using the three places of storing data. For the IoF application a similar approach to the shared memory place was chosen.

The standard way to implement this application would be to read from the database each time a user wants to plot something, but when a user for example want to plot multiple tags for the whole time period, it will take quite some time to read all the data from the database and then transform it to the appropriate format. This was tested, and it took over a minute to retrieve a graph for large amounts of data, which is not desirable. NB, it is important to note that it was discovered late in the development that the definite slowest part of this process is conversion of datetime-objects to string inside `pandas DataFrames`. It would be beneficial to store the datetime as a string when receiving new messages to avoid this problem. Besides transforming timestamps to datetime, and then transforming datetime to string, there was no particular need for post-processing, except setting the column data types to `uint8`, `uint16` and so forth for better memory optimization. Still, the speed problem of reading from database will only increase as the data source keeps growing, but it could perhaps perform a lot better than previously assumed.

To get around the issue, a Apache Pyarrow Plasma store was used [2]. Apache Arrow is a cross-language development platform for in-memory data, where `pyarrow` is the Python-specific implementation. Plasma is an in-memory object store and is a part of Apache Arrow, holding immutable objects in shared memory. The flow of the Dash application implemented was as follows:

First a process is started to allocate memory to a Plasma store that is kept running. Second a start-up script is started that reads all current data from database, performs post-processing and transformation to the appropriate `pandas DataFrame` format, and then sends the dataframes to the plasma store. The ID to retrieve them again from the store is then stored in a `pickle`-file on disk. Finally the Dash application is started. Every time a graph is to be updated, the complete dataframe of the chosen data set is retrieved

from the plasma store and then filtered appropriately. A separate interval component in the Dash layout triggers every n seconds and polls the database for any new messages. If there are any, it retrieves new messages from database and perform post-processing. Then it reads the complete dataframe from the plasma store, appends the new data, and sends the dataframe back to shared memory.

By serializing and deserializing the data this way it is assured that all threads and workers have access to the most current version of the data, already processed for faster expedition. This approach greatly improved the Dash app, reducing the processing time from over a minute down to a couple of seconds.

Performance considerations

A default Plotly scatter plot will use vector graphics (SVG) to render the graphics of the plot. A limitation with SVG is increased rendering overhead as number of elements increase. This is a limitation with all interactive web visualizations using vector graphics for large amounts of data. While one might be able to load the inital plot, interactivity will be sluggish and hard to use. Plotly state that they have a hard time with more than 500k data points for line charts, or 40k points for other types of charts [31]. In the IoF use case, one might wish to see the lifetime of a tag in a single scatter plot, and likely more than just one tag at the time.

Luckily Plotly also support WebGl, a Javascript web API made to render 2D and 3D graphics in web browsers. WebGl uses the GPU to render graphics instead. Using WebGl to make scatter plots, or other supported plot types, we increase speed, interactivity, and amount of data greatly. WebGL can easily plot 1 million points in scatter plot at once, or multiple line traces with large amounts of data. The 2D and 3D scatter plots employed in the Dash application is utilizing the WebGl counterpart for these reasons. The drawback is that the graph will be rasterized should one wish to export the graph to an image format.

As mentioned, sharing data between callbacks is one of the major challenges in Dash. An in-memory object store like the Plasma store used made in the IoF app improves performance of Dash apps in terms of speed. However, it will clearly decrease RAM performance. A solution to increase speed without the cost of large in-memory stores is

file-system memoization caching. Setting a function be cache memoized, it will return the same output for an input it has received before. One could for example store the full project post-processed data in a file-system cache every 24 hours. A callback updating the front-end figure would then fetch all previous data (complete cache), and then only read new data since the caching point from database, post-process it, and append to the existing dataframe. Caching is not as speed-efficient as the RAM-based Plasma solution, and the total amount of processing is increased. Nevertheless, it can be a good method to improve performance, especially if RAM is limited.

Construction of the case-study app

Figure 4.17 shows the implementation of the Dash IoF application. It was mainly developed for the case-study, and is therefore not directly applicable to other experiments. However, generalizing it should not be too difficult. Functionality was prioritized over design and optimization, and so the app supports several options and customizations. Since Dash is designed to abstract away web development, it is fairly simple to apply styling and structure of the webpage. In this case, the design was based on `css`-files from the `dash-bio` component [32]. In regards to security, Plotly offers robust authentication ready-made for Dash for commercial users, but there exists a multitude of open-source authentication options for Flask apps, that can also be used for Dash. Basic authentication was implemented in the IoF app through the free and basic `dash-auth` plugin. Figure 4.16 shows a simple icon designed for the IoF webpage that illustrates the IoF concept.

The webpage is designed to give the user three tabs with a control interface on the left-hand side: data, plot, and filter. On the right-hand side a large frame for the graph is placed. The elements of the page are declared as separate html <**div**> components, which can be rendered invisible or visible at any time by accessing the `display` property. The underlying layout code was written so that whenever the user switches datasets in the app, the currently rendered components would be turned invisible, and rendering newly selected components visible. Data tab remains the same, but the plot and filter tabs are interchangeable, as well as the graph container. In addition there is an *About* tab on the far left side of the controls that is static, and a *Submit Changes* button below the controls. The

Figure 4.16: IoF illustration icon. Made as 'branding' of the IoF concept, used in webpage header.

button is also unique for each dataset as separate html components, and it too is rendered invisible depending on the current dataset.

So each dataset has a group of components tied to them, that are rendered in the webpage or invisible when changing dataset. Each component therefore also have separate callback functions, although some of them will be direct duplicates of each other, only the component they return to will change. The submit button for each dataset is the only Dash `Input` to all three graph figure callbacks. This way, the active graph is only updated whenever the user wants it to, by pressing the button. This reduces delay and performance issues greatly compared to changing the graph on every interface input change. Another benefit to this approach is that only one graph is handled each the button is pressed, instead of performing a callback for each graph even though only one of them is visible. The other callbacks of the app is fed as `State` to the graph callbacks, but `Input` in their own callbacks, making other changes in the app instantaneous, like switching values in a dropdown menu.

The alternative approach would be to have only one html component for each part of the

webpage, and use internal logic in the callbacks to change the contents of the components (i.e. calculating and returning a tag graph by checking if tag is the currently selected dataset). This reduces duplication of code, but increases complexity of each callback. Another approach would be to render all of them at the same time, but this decreases performance, especially if each graph is displaying a complex plot. Additionally, one could separate the different datasets in separate Dash apps. By importing shared callbacks are regular functions, duplication can be avoided, and each app would be less interconnected and complex. However, the current design was chosen to allow the user to freely switch between datasets within the same webpage, while also remembering plot configurations when switching back, at least in the same active session.

In the data tab, the user can switch between tag, tbr, and positional data. Switching from tag to TBR would make the tag graph, submit button, and plot and filter tabs, inactive and invisible. TBR graph, submit button, and plot and filter tabs, would immediately be rendered active. This design allows Dash applications to remember configurations for a dataset. Also, the changes happen in the background, so the user would not notice anything going on.

In the plot tab, the user could select which type of plot they wanted, which data they wanted on the axes, and customizations of some plot types. These customizations were marker size, line width, and line type (solid, dot, dash, etc.) for scatter and line plots. For boxplots one could choose whether to group boxplots or overlay them. By using the aforementioned callbacks, all of these changes were connected, interactive and instantaneous. To illustrate the great usability of this, the available data types for plot axes were automatically changed when switching datasets (temperature only available for TBR for example). In addition, axis options and customization options were disabled if not valid for the chosen parameters. Choosing z-axis data was for example only possible for 3D plots.

The filter tab was used to filter the current data, by setting options that were later used in the graph callback. Specifically, the callback would use the filter parameters as `State` and apply them to the dataset dataframe, before creating and returning the figure object. All datasets shared some filters, like a start date to end date filter and a time of day filter.

The tag dataset had a unique SNR filter option as a slider, and shared tag ID selection filters with the positional data. TBR had unique filters for average and peak ambient noise, as well as temperature, all implemented as sliders. By "shared" filters it is implied identical filters, but each had a unique copy when running the Dash code, operating individually of each other.

Separate real-time monitoring app

Taking the main IoF app as a starting point, a separate simplified app showing tag data from the last 24 hours was also made to support real-time monitoring of fish. Only basic scatter plot was available in the app, and only tag detection data, but the customizations were available, like choosing to have markers, markers+lines, or only lines. All the filter options too. However, in this app, where the focus was to monitor fish in real-time, there was no button to update the graph. Instead, an interval component was triggered every n seconds to update the figure.

In this application, the Apache arrow plasma in-memory object stored was not used. Instead, cache memoization was used to store post-processed data from the previous 24 hours in file-system cache every hour. When the interval callback function was triggered, it would read from cache by calling a read-data function with the current hour as input. Because of memoization, the function would immediately return the post-processed dataframe from file-system. The callback proceeded by calling a similar read-data function that retrieved all data from start of this hour until current time. The callback combined the dataframes, created a figure, and returned the updated figure. Notice that the new data is not saved in any capacity using this solution. Saving would defeat the purpose of caching, since the read-data function would be called with unique inputs each time, making it impossible to retrieve the same cache. With the chosen set-up, the app would at most have to post-process 59 minutes and 59 seconds of data. When calling the memoized function with a new unique hour, the function would overwrite the cache of the same hour the previous day.

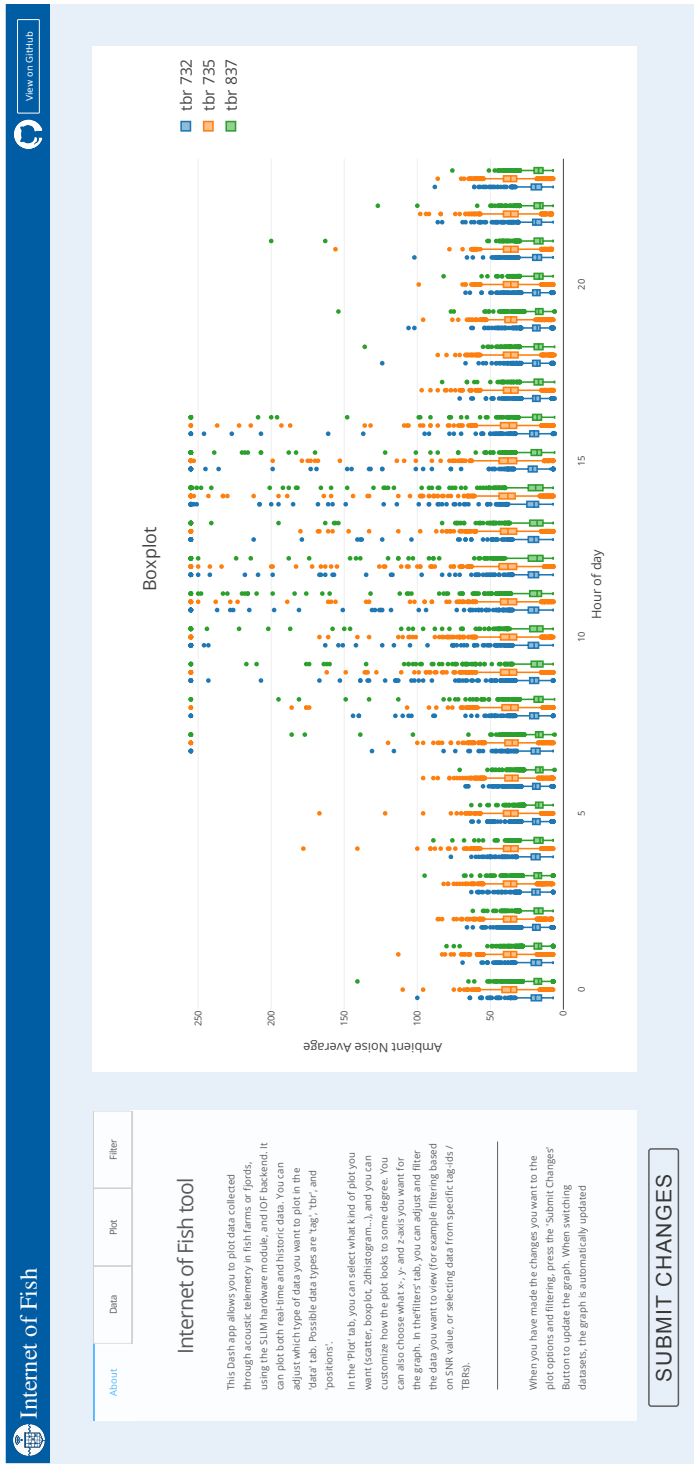


Figure 4.17: IoF front-end Dash app design. This is how the webpage design looks like. Some navigation tabs on the left, where it is possible to select filters etc that affect the current graph to the right. This is a cropped view of the webpage (only whitespace has been removed). It adapts to different displays, but no other displays or resolutions than the development computer was tested during development. The app was developed on a 16:10 display with 1900x1200 resolution.

Chapter 5

Case study - Aqautraz

To validate the IoF concept, a use case and test was planned and conducted. Work has been done in cooperation with Midt-Norsk Havbruk AS on their site positioned in Nærøy, Eiterfjorden, north-east of Kolvereid. They have developed a new sea cage concept called Aquatraz [28]. The Aquatraz system is designed as a solution to the problem of sea lice and fish escaping cages. The first meters below water are made of rigid steel (a so-called lice-skirt), with windows in the frame below 8 meters (fig. 5.1). The water inside the cage is pumped up from deeper underneath, where the lice is less prominent, and circulated down and out of the cage through the net further down in the cage. The whole cage can be raised and lowered on the water surface, potentially easing the process of removing fish that should be harvested and general operation. The cage is 160 meters in circumference, 8 meters steel skirt, 18 meters net depth, with the lowest net point 32 meters down (when lowered). When raised, highest structure of the Aquatraz cage is 21.2 meters above the water level (4.6 meters when lowered).

The IoF system was instrumented at Aquatraz the 5th and 6th of November as part of co-supervisor Waseem Hassan's work [17] and for the specialization project prior to this thesis work [19]. The Aquatraz cage and a normal sea cage was instrumented with 60 acoustic tags, 30 in each. 40 of the tags (20 in each) alternated between sending depth and acceleration. They transmitted approximately every 60-90 seconds, and had a battery life of around 9 months. They were sending with frequency 69 KHz. 20 of the tags (10

in each) sent only depth, every 20-40 seconds, on two separate frequencies different from the alternating tags. They were sending with frequency 71 and 73 KHz. 3 SLIMs and 3 LAMs with a TBR each were placed in the Aquatraz cage, and 3 SLIMs with 3 TBRs were placed in the reference cage. LAM stands for LoRa Add-on Module, which is a separate hardware for IoF which the SLIM module is based on. The LAMs were only there for redundancy as the SLIM module had not been tested in a full-scale experiment yet. The cage has metal spokes across that were practical to use for attaching the SLIMs and to lower the hydrophones some distance away from the cage walls. Figures 5.4 and 5.5 show images from the instrumentation work, the sea cages, and the equipment used.

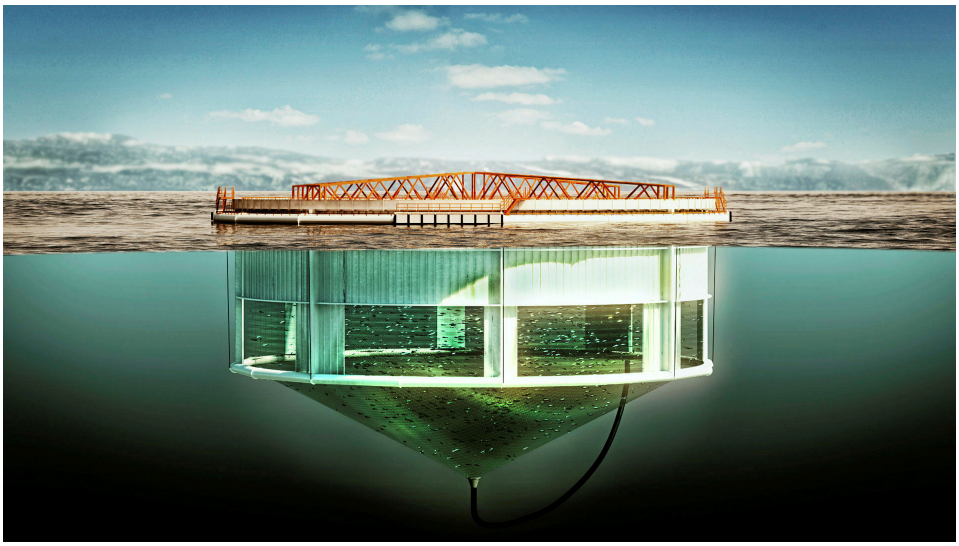
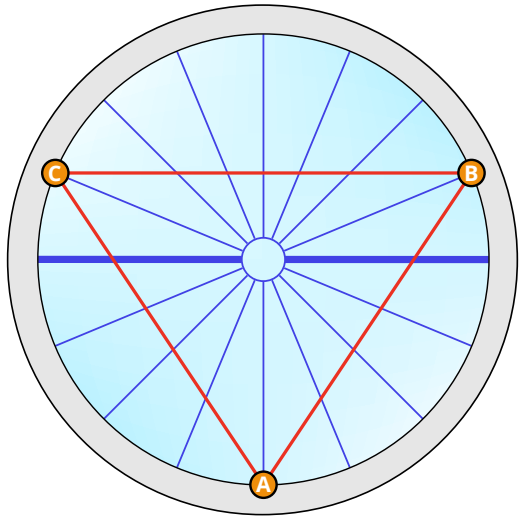


Figure 5.1: The Aqautraz sea cage concept by Midt Norsk Havbruk AS and Seafarming Systems AS. Reprinted from [28]. Copyright of image belongs to Seafarming Systems.

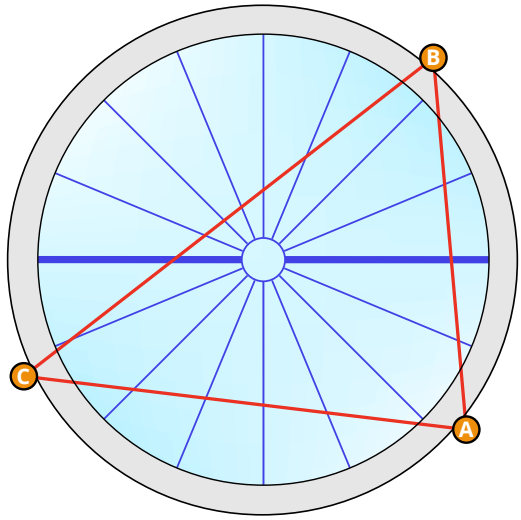
The initial instrumentation was using old software with the old IoF protocol, sending each packet with fixed 11 bytes. The old LAM modules and the gateway was removed from the site early 2019. This was to investigate issues with the gateway, the SLIM modules were functioning fine, so the LAMs were no longer needed. 10th of May the gateway was brought back to the site, and set online again. The SLIM modules in both cages was also updated with new software supporting the new message protocol. And the SLIM-TBR pairs in Aquatraz was moved outside the cage. This was because Midt-Norsk Havbruk were

set to lift the cage 20th of may. If they had been left in their original position, the TBRs would be hanging in the air high above the water, loosing valuable data of fish behaviour when lifting the cage frame.

Figure 5.2 shows how the SLIMs and hydrophones were setup on the Aquatraz cage, where fig. 5.2a shows the original set-up, and fig. 5.2b shows the set-up after 10th of may. The triangles in fig. 5.2 shows the area of high accuracy for positioning, originally close to a unilateral triangle. Figure 5.3 shows how the cages are positioned in relation to each other and the feeding barge (not to scale). In the reference cage, there are 60 support beams (fig. 5.5) around the cage. Three SLIMs with a TBR each were placed with 20 support beams between each.



(a)



(b)

Figure 5.2: Setup of SLIMs and TBRs on Aquatraz farm. The thick horizontal blue line in the middle, and the gray area around the water, shows where it is possible to walk. The blue lines leading to the middle are metal spokes, which were used to attach and lower down the TBRs from in 1, 2 and 3. The cardinal directions are for reference, where the feeding barge is located approximately directly south. The triangle illustrates area of most reliable positioning.

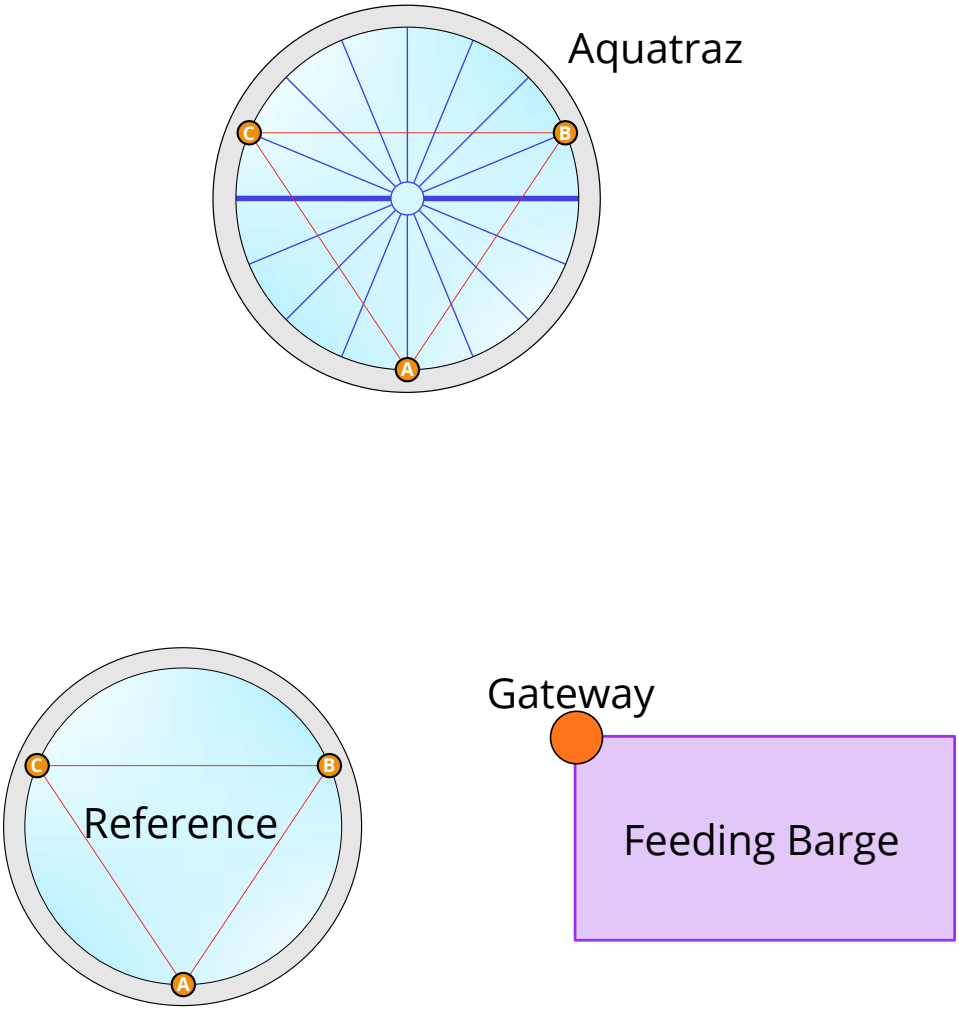
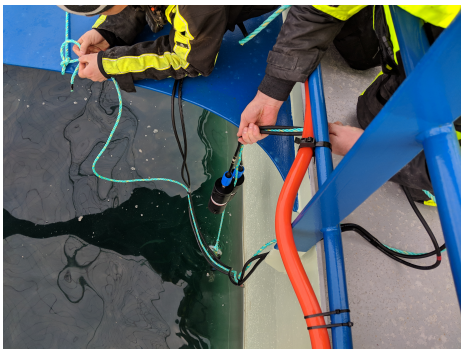


Figure 5.3: Setup of SLIMs and TBRs on Aquatraz farm. The thick horizontal blue line in the middle, and the gray area around the water, shows where it is possible to walk. The blue lines leading to the middle are metal spokes, which were used to attach and lower down the TBRs from in 1, 2 and 3. The cardinal directions are for reference, where the feeding barge is located approximately directly south. The triangle illustrates area of most reliable positioning.

(a)

(b)

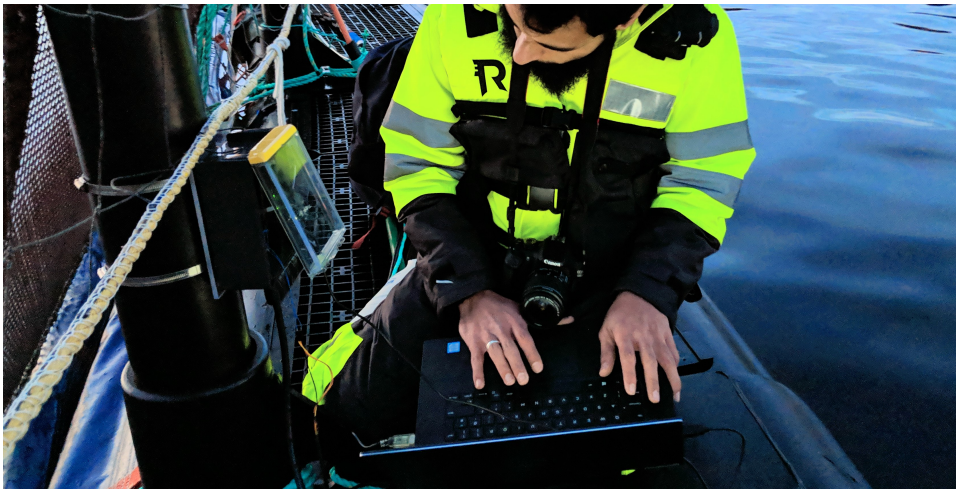


(c)

(d)

Figure 5.4: **(a)** Panorama image of Aquatraz sea cage. **(b)** Old LAM module to the left, new SLIM module to the right, connected to a TBR each. **(c)** TBRs being lowered into Aquatraz cage, about 3 meters deep. **(d)** Acoustic tags from Thelma BioTel used for instrumenting fish in both Aquatraz and reference cage. Photos taken by Per Arne Kjelsvik, the author of this paper.

(a)



(b)

(c)

(d)

Figure 5.5: **(a)** Panorama image of reference sea cage. **(b)** Debugging on the edge. **(c)** Barge on-site as seen from the reference farm. Gateway placed behind window on left corner wall visible in the image. **(d)** Acoustic tag being inserted into a salmon. Photos taken by Per Arne Kjelsvik, the author of this paper.

Chapter 6

Results

In this chapter, results for implementing IoF back-end and front-end is presented. Both parts of the application are validated following the lists of requirements defined in chapter 3. Section 6.3 presents the various plot types that the Dash app supported using case study data. Section 6.4 shows more plots from the case study to further illustrate visualization and showcase data from the case study. All figures are generated in the Dash application, or through a simplified command-line interface. In both cases they are exported as vector graphics, except the 3D scatter plots which contain rasterized scatter traces, and the 2D histogram plots without contours. This is a limitation for these plot types in Plotly.

No measures have been taken to eliminate underlying factors in the dataset for the presentation of data from the case study. This should be kept in mind when assessing what the data means. The data these figures are based on was collected from the various TBRs manually, not through the IoF system. This means that all available data from the case study has been used. This was a natural choice, as the gateway was not on site February - May, and was offline intermittently after 10th of may before going completely offline 10th of June. In all figures, the data used is from 6th of November until 10th of May, unless otherwise stated in the figure text.

6.1 Back-end software

The back-end software was written using Python `3.7.3` and was run in a Python virtual environment on a `Linux Ubuntu Server 18.04 LTS` virtual server. The server ran the main back-end program with a shell script, ensuring that operation would resume if problems such as power outages cropped up. No apparent issues with the back-end client surfaced while receiving messages, both when testing in-office and live from the case study.

The back-end successfully unpacked incoming MQTT messages, and subsequently IoF messages, before inserting them to a database. The incoming MQTT messages were `JSON` strings, in the following format:

```
mqttMessage = {'data': base64-byte-string, 'snr': float}
```

The `'data'` in this context is an IoF message, and `'snr'` is a SNR value for the LoRa transmission of said IoF message. The client unpacked the data, decoded the base64 string to get the IoF bytes message, processed the bytes to retrieve IoF data, and inserted the results to database. When finished, the back-end inserted the original base64-string and snr-value to a backup database. The SNR values of the LoRa transmissions have not been used or analyzed in any capacity during the project period. If unrecoverable problems arose while processing an IoF message, the back-end would not attempt to insert the message to the main database, and when inserting backup of the mqtt message, it would not include a message ID for that message. This way, all successful messages in the main database could easily be found in the backup database to inspect the SNR value or look through the IoF message manually.

A fully working logging system was implemented for the back-end client using the built-in Python `logging` package. The logger was set-up as a timed rotating file handler logger. Midnight each day, the software would create a new log-file in the back-end file path. This way log files did not grow exponentially in size as time passed on. Every time a message would fail in unpacking, the software would log the full message, the interval `rowID` of the message in backup database, and an accompanying error message. By using typical `try-except-else` patterns, the software was able to maintain operation when faced with errors, and at the same time provide insightful error messages. The logger also

included `traceback` information for critical exceptions. There were not observed many errors per day running back-end, and all errors that did happen, happened as a result of corrupt messages. An example of this would be a message with an invalid codetype value, making unpacking impossible.

The main `pip`-packages used for the development of the IoF back-end are listed below. They can all be installed through the `pip` package manager. A specific `NumPy` version is installed when installing `pandas`, as it is built on `NumPy`.

- `paho-mqtt 1.4.0`
- `pandas 0.24.2`
  - `numpy 1.16.4`
- `toml 0.10.0`
- `utm 0.4.2`

6.1.1 Validation of back-end

In this section, the requirements defined for back-end in section 3.4.1 are used to validate the back-end software. To which degree the requirements were met is addressed in the following lists. The shared requirements (section 3.4.3) for back-end and front-end are validated in section 6.2.2.

Functional back-end requirements (FRs)

- **FR1 -** back-end used `paho-mqtt` as MQTT client, an established and widely adopted MQTT client for Python .
  - **FR1.1:** Back-end was able to subscribe to specific project-specific topics, but in the implementation of the case study app, the app was set to subscribe to the wildcard topic # for simplicity. No method to feed topics through an interface or file was implemented in the back-end, so any specific topics would have to be hardcoded in.
  - **FR1.2:** The chosen MQTT broker, `mosquitto`, handled authentication set-up such as SSL and username/password. The client enabled these options, and so

SSL was used with an accompanying username and password. The gateway in the network layer was not using SSL for mqtt communication.

- **FR1.3:** `paho-mqtt` fully supports MQTT version 3.1.1. At time of writing, only the C library version supports the new MQTT version 5.0.

- **FR2 -** IoF message parsing was successfully implemented as a sub-package in the back-end source code.
  - **FR2.1:** The sub-package used the protocol format defined in chapter 4 to parse messages.
  - **FR2.2:** By using `toml`-metadata files, the back-end successfully converted raw data while unpacking IoF messages.
- **FR3 -** Back-end stored raw, converted and positional data to an `SQLite`-database file.
- **FR4 -** Back-end solved TDOA for incoming messages to find positions.

Non-Functional back-end requirements (NFRs)

- **NFR1 -** The main run program spawned the client, which is the core of the back-end, in a subprocess. If the subprocess died, the main program would respawn the process. In addition, the main program was run as with a shell script on a virtual server, ensuring that it would restart should a power outage or similar happen.
- **NFR2 -** Backup of data was stored in a separate database than the main IoF database. All messages was stored in backup, but if message parsing or insertion to database failed, the backup insertion would not include the message ID of the message.
- **NFR3 -** `SQLite` was chosen for the persistent storage, a reasonable, flexible, and easily maintainable solution.
  - **NFR3.1:** Structure and format of persistent IoF data defined in section 4.2.1.
  - **NFR3.2:** Storing raw bytes of the message is only done in backup, and storage in `SQLite` database files ensures a widely adopted storage solution.
- **NFR4 -** All code in back-end was implemented with solid `try-except-else` patterns.

- **NFR4.1:** Errors were logged with Python 's built-in logging package. Errors were logged in separate log-files for each day that changed to the next day at midnight.
- **NFR4.2:** The logging package supports 5 levels of logging: 'debug', 'info', 'warning', 'error', and 'critical'. Only level 'warning' to 'critical' was included in log files by default. All levels were printed to terminal by default.
- **NFR4.3:** Using the `try-except-else` blocks, corrupt and invalid IoF messages were detected, and handled appropriately. Errors were written to log, and information was printed to terminal. Normal operation of back-end was unaffected by this.

- **NFR5 -** Metadata file-system file handling was successfully implemented in the various modules of the back-end software.
  - **NFR5.1:** A standard format and structured was defined using `toml`-files.
  - **NFR5.2:** Conversion of data was done in message handling, and it was subsequently stored in database along raw data.
- **NFR6 -** TDOA positioning was implemented in back-end (Appendix A).
  - **NFR6.1:** Back-end initialization would create a separate metadata `toml`-file from the main metadata file, that could be used for positioning. In this file, parameters such as cage radius, center coordinates, TBR locations was saved. Thus the positioning module could use this to position tag detection triplets.
  - **NFR6.2:** Positional data was stored persistently with XYZ coordinates.
  - **NFR6.3:** Positional data was stored persistently with latitude-longitude coordinates. Back-end would first convert latitude-longitude positions of TBR to UTM, and then use UTM to create a local coordinate system. When done, the coordinates would be rotated and translated back to UTM, and then converted to latitude-longitude. The simple `pip`-package `utm 0.4.2` was used for the conversion between UTM and latitude-longitude.
  - **NFR6.4:** A method to find triplets fast and efficiently in a complete database set was developed, see section 4.5.
  - **NFR6.5:** A method to adjust triplets timestamps fast and efficiently was de-

veloped, see section 4.5.

- **NFR6.6:** Positioning was performed on each message after it was successfully handled and stored in database.
- **NFR6.7:** Positioning on complete IoF database was also implemented successfully.
- **NFR6.8:** Code was written so that one could use the aforementioned metadata file, or one could use the database GPS information from SLIMs to determine TBR locations, and not use cage radius and center (no resolving position with distance from cage center). The changes did however require the developer to manually go in and change source. No fluent system to automatically switch over to simpler sources if no metadata was present.

- **NFR7 -** Conversion of raw Thelma BioTel TBR databases was implemented and used to convert their format to the IoF format. However, this code can not be shared, as the structure and inner workings of Thelma BioTel databases are proprietary.

UI back-end requirements (UIRs)

- **UIR1 -** Command-line interface to setup MQTT connection was implemented.
- **UIR2 -** Command-line interface to initialize new project was implemented.
- **UIR3 -** Command-line interface to reset and/or delete current project was implemented.
- **UIR4 -** No GUI or command-line interface was implemented to enable or disable varying levels of interface printing / error log writing. It could change manually in two locations of the code, and a simple command-line option would not be hard to be implement, but nothing of the sort has currently been implemented.

6.2 Front-end application

The application was developed and run on a `linux Ubuntu 18.04 LTS` computer that had 32 gigabytes of RAM. The application was written using `dash v0.43.0` in Python `3.7.3`. A virtual environment using the default `venv` was created to contain code libraries.

The display screen used to develop the application was a 16:10 1900x1200 display, and this was also the only resolution the app was tested for. It should scale to different screens, but smaller resolutions will not maintain the same design, like phone screens, due to limitations in resolution size.

The application successfully supported multiple users at once using controls to customize graphs to analyze, connecting from remote computers. The application was tested by running the app locally on the development computer. Remote access was given by opening up the application ip-port to the complete NTNU subnet. No thorough tests were done to check performance with multiple users, but serving three users at once did not raise any apparent issues. As detailed in section 4.5.2, basic authentication was implemented with the `dash-auth` plugin, and the design of the app was based on the `dash-bio` component library. The app supported multiple plot types, such as scatter-plot, line-plot, box-plot, histogram, 2D histogram, 2D histogram contour, 3D scatter, and animation of 3D scatter.

A separate simplified app showing data from the last 24 hours was also made to support real-time monitoring of fish, where the application automatically polled the database for new data every 10 seconds, and updated the graph without user interaction if new data was present. This app only supported basic scatter and line plots. There were no apparent issues with the real-time application, but extensive testing could however not be performed, since the gateway went offline intermittently. Eventually the gateway went offline permanently, making further testing impossible.

The main `pip`-packages used for the development of the IoF app is listed below. They can all be installed through the `pip` package manager. The nested lists indicate the most important dependencies this version of the package installs. The other dependencies of the packages are not included for brevity.

- `dash 0.43.0`
  - `dash-core-components 0.48.0`
  - `dash-html-components 0.16.0`
  - `dash-renderer 0.24.0`
  - `dash-table 3.7.0`
  - `Flask 1.0.3`

    - plotly 3.10.0
- dash-auth 1.3.2
- pandas 0.24.2
    - numpy 1.16.4
- pyarrow 0.13.0

The app supported offline download of multiple figure formats, but only to the local computer it was running on. This requires either the `plotly-orca` package (not available through `pip`), or by installing `psutil` and using `plotly` API to export figures. A modified copy of the app was used to generate figures, where some options were changed to increase printed readability. For this version `psutil 5.6.3` and `plotly` API was used, which can be installed through `pip`. The download option was implemented using a dropdown menu and a button. This export option ignores any user modification to the active graph, such as zooming or panning. Normal dash figures support download of graphs as compressed png files, but in the modified version, that option was changed to SVG. This method only attainable for files smaller than 2 megabytes, meaning large vectorized scatter traces could not be downloaded. Nevertheless, this option allowed the user to include modifications, such as zooming, before saving the vectorized figure.

6.2.1 Validation of front-end

Functional front-end requirements (FRs)

- **FR1 -** Front-end was implemented as a Dash app, a library that supports a wide range of browsers by default. No extensive testing was performed, but no issues were experienced in Google Chrome, Opera, Firefox, or Safari. No additional installations was required to use the application.
- **FR2 -** A separate simplified app was created for live updating real-time data graph. It was operational for around 1 week before gateway went offline. During that time, the app worked as intended.
- **FR3 -** The menus was instantaneous to use, while the plotting would take below 10 seconds for the simplest graph, around 10-20 seconds for more involved figures, and around 30 seconds for the most involved plot (3D animation).

- **FR4 -** The implemented application was not project agnostic. It was custom tailored to the case study, and would not work with any other projects. The majority of the app could however be reused in other projects, requiring some refactoring of code.
  - **FR4.1:** No documentation for setup of front-end was written.
  - **FR4.2:** No metadata files were used, and so no format or procedure on how to set-up metadata was defined.
- **FR5 -** The application supported a wide variety of basic, scientific, statistical, and 3D charts. Section 6.3 showcases all plot types that the app supported.
  - **FR5.1:** Scatter and line plots was implemented (fig. 6.1), filled area was not.
  - **FR5.2:** Boxplots (fig. 6.4), horizontal (fig. 6.3) vertical histograms (fig. 6.2), and 2D histograms (fig. 6.5) were implemented.
  - **FR5.3:** Contour plots was implemented (fig. 6.6). The heatmap plot type that `plotly` provides was not implemented.
  - **FR5.4:** 3D scatter was implemented (fig. 6.7), and for positional data, 3D animation was implemented (fig. 6.8). Surface plots were not implemented.
  - **FR5.5:** Timeseries plot was implemented (fig. 6.1). The user could simply choose 'Date' as the x-axis, and it would be a timeseries plot. A button was also implemented that would create a rangeslider and navigation buttons to simply timeseries plot navigation.
- **FR6 -** The app supported plot customization with controls on the left-hand side of the application (fig. 4.17). The user had options to customize the look of scatter and line plots, and boxplots. There was no options implemented to select colors of the plots, or speciality features such as choosing custom bin sizes for histograms.
  - **FR6.1:** Dash provides several tools for zooming, panning, and other controls by default in all figures.
  - **FR6.2:** Tools for zooming, panning, and selecting data etc.
  - **FR6.3:** The user could freely choose x-, y-, and z-axis.
  - **FR6.4:** The user could not change axis labels, plot titles, or colors. Functionality to adjust marker size and opacity, line width, and line style was however implemented.

- **FR7 -** The user could freely filter the different datasets with multiple inputs.

Non-Functional front-end requirements (NFRs)

- **NFR1 -** No steps was taken to deploy and host the application. The application was run on a local computer, opening up access to the port of the app to specific computers. For testing, the app was opened up to the subnet of IP-addresses belonging to NTNU.
- **NFR2 -** Basic authentication was implemented with `dash-auth`, requiring a username/password present in a file on the local computer running app.
- **NFR3 -** No table view was implemented.
- **NFR4 -** No data export options was implemented.
- **NFR5 -** Export of plots was implemented, but only to the local computer running the application. Dash supports download of compressed `png`-files of any plot by default.
- **NFR6 - Export of plots:** in multiple formats and configurations.
  - **NFR6.1:** The host download of figures supported pdf, eps, svg, webpage, and jpeg.
  - **NFR6.2:** Figure sizing could not be customized, and the download option would also disregard any interactive changes to the figure. It would only download the first view of the graph that the user was served. A Python script was written using code from the app to export figures manually. This application has been used to create some of the figures of this chapter.
- **NFR7 -** The app does not have any option for end-users to provide their own datasets.
  - **NFR7.1:** No option to upload dataset, so no option to remember user datasets in-between sessions either.

UI front-end requirements (UIRs)

- **UIR1 -** No steps were taken to ensure proper dynamic sizing of webpage to different devices. No testing was done for other displays than the development display (1900x1200).

- **UIR2 -** No app gallery was created, but a separate application for real-time was implemented.
- **UIR3 -** No dashboard functionality was made for the front-end.
- **UIR4 -** Controls of the app was clearly labelled and intuitive to use.
- **UIR5 -** Navigation / switching between data sets was done with a single dropdown menu.

6.2.2 Validation of shared requirements

- **NFR1 -** All code was written in Python with minimal dependencies.
  - **NFR1.1:** Both back-end and front-end relied on the same version of `pandas` and `numpy`. Front-end relied on three more libraries, `dash`, `dash-auth`, and `pyarrow`. Back-end also relied on three more libraries, `paho-mqtt`, `toml`, and `utm`.
  - **NFR1.2:** All code written in Python , with the exception of `css`-files used for webpage layout.
- **NFR2 -** Python `3.7.3` was used for both back-end and front-end. At the time of writing, this is the newest stable full release of Python . Version `3.7.3` was released 25th of March 2019, while Python `3.7.0` was released 27th of June 2018 [[35]].
  - **NFR2.1:** `dash 1.0.0` was released shortly after development of application was finished, which formalizes the developers promise to not introduce any breaking changes until the next major version release. All other tools and libraries are fairly established and stable.
  - **NFR2.2:** None of the chosen libraries and tools have announced plans to stop development in the near future. Most are relatively modern tools.
- **NFR3 -** html documentation was produced with the `pip`-package `sphinx`. However, the documentation was incomplete as not all parts of the back-end was documented, and none of the front-end code was documented beyond simple inline comments.
  - **NFR3.1:** Documentation in back-end followed for the most part the Google

styleguide for documentation. Most of the code was well-documented, both with `docstring` comments, and type hinting. Front-end was not documented at all, beyond sensible function and variable naming.

- **NFR3.2:** Documentation generated with `sphinx` is included in the source code as a html webpage.

- **NFR4 -** Both back-end and front-end needed access to the same database, but only back-end needed to write to the database. No other dependencies between them.
- **NFR5 -** All libraries and code used is open-source. Only proprietary parts was in the form of extra code to translate Thelma BioTel databases to IoF database.

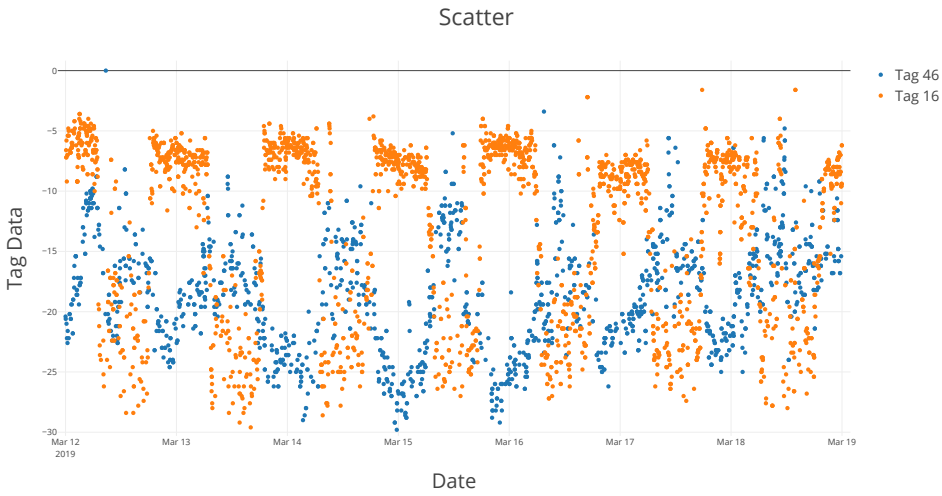
  - **NFR5.1:** IoF is available in an open-source repository [20]. At time of writing, the front-end application cannot be run, as it depends heavily on the case study.
  - **NFR5.2:** All code used is open-source.
  - **NFR5.3:** No proprietary libraries or tools used.

6.3 Front-end plot types

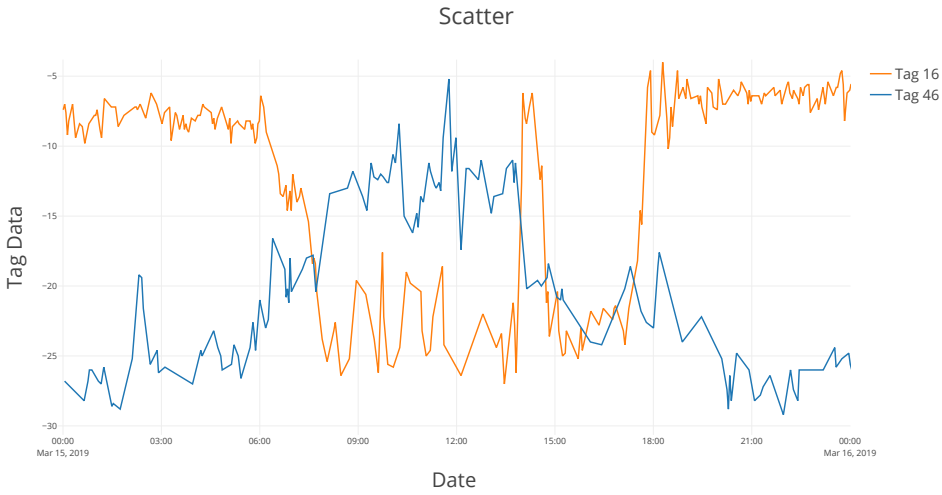
In this section, the various plot types supported by the Dash application are presented. Some plots of each type the app supports are presented. Figures 6.1 to 6.8 show examples of all plot types supported in the front-end application. Section 6.4 presents additional figures showcasing data from the case study.

Figure 6.1 show examples of scatter and line plots in Dash, in this case the depth of two fish in Aquatraz 1 week in march (12.03.2019 - 19.03.2019). A similar pattern such as the one these two exhibited was observed in most of the fish, namely a pattern where the fish stay at a certain depth for most of the day, before swimming to another depth, and then returning to the same depth. Regardless of which way the fish swam, the time window they fish swam around was seen to be from 7 to 16 in the start of the project period. However, as the case study progressed, this window of activity was gradually shifted to a window of activity closer to 6 in the morning and 18 in the afternoon.

Figure 6.2 shows examples of vertical overlaid histograms in Dash, in this case displaying the SNR value of all tag detections in both case study cages. Figure 6.2 shows the



(a)



(b)

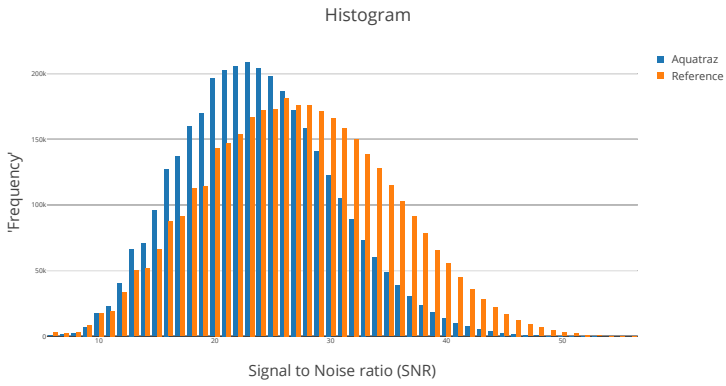
Figure 6.1: Scatter and line plots. **(a)** Scatter plot for the depth of two fish 1 week in march. **(b)** Line plot showing a zoomed in view (March 15th) of the same chart.

total SNR, fig. 6.2b shows SNR from 7 in the morning to 16 in the afternoon, and fig. 6.2c shows SNR from 16 in the afternoon until 7 in the morning. Aquatraz was seen to have worse SNR values, with the total histogram centered around 23, while reference cage was centered around 26. However, take note of how reference SNR varied in figs. 6.2b and 6.2c.

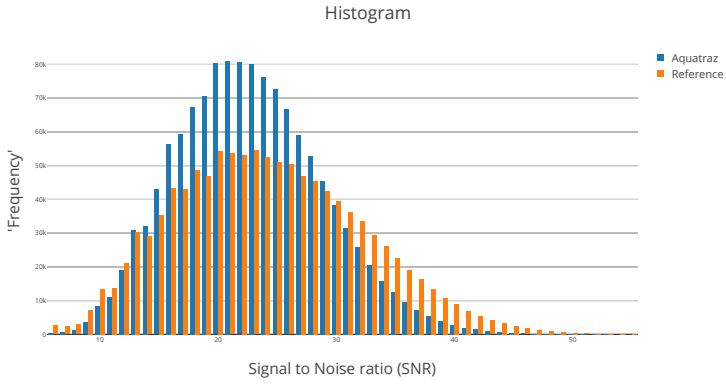
Figure 6.3 shows examples of non-overlaid horizontal histograms in Dash, in this case the depth distribution of three tags in Aquatraz. As can be seen in the histograms, the three tags were observed with distinct depth distributions. In the data from the beginning of project period (first month), several fish in Aquatraz were observed to be distributed around two depths, more aligned to a bimodal distribution than a unimodal one. Figure 6.3b shows some resemblances to a distribution like this, but the lower peak is clearly not as well-defined. In reference cage no tags were observed have a bimodal distribution. Reference tags were also observed to have varying depth distributions.

Figure 6.4 shows examples of boxplots in Dash, in this case boxplots of TBR data in both Aquatraz (fig. 6.4a) and reference (fig. 6.4b). The boxplots show how ambient noise average measurements vary with the hour of day. Notice that one TBR in Aquatraz, TBR 735, had a consistently higher boxplot noise average than the other two TBRs, and with slightly larger standard deviation too. In reference, one can see that two TBRs (730 and 734) had higher noise averages, and a greatly increased standard deviation from 8 until 16 in the day. The last TBR in reference, TBR 836, can also been seen to have had increased noise average in these hours, but much less pronounced. Figures 6.10 and 6.11 in the next section show a more detailed view of the same boxplots.

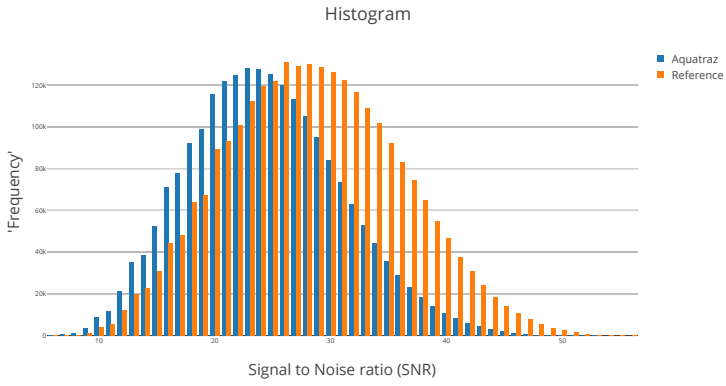
Figure 6.5 shows examples of 2D Histograms in Dash, in this case 2D histograms of time of day and SNR value in both case study cages. This plot type is at the time of writing not possible to export from Dash or plotly in vector graphics format. In fig. 6.5a the SNR and time of day relation in Aquatraz, fig. 6.5b in reference. From the plots, it can be perceived that Aquatraz had a less varying SNR value, with only a slightly degradation during midday. Reference had on the other hand a wider span of values centered around a higher SNR value, with a much more drastic degradation in SNR quality in the midday below Aquatraz in the same day window. A significant difference in intensity can also be observed between midday and the rest of a day for the reference cage.



(a)

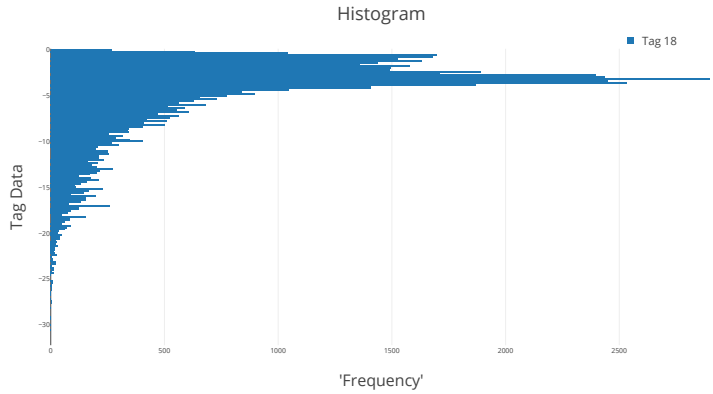


(b)

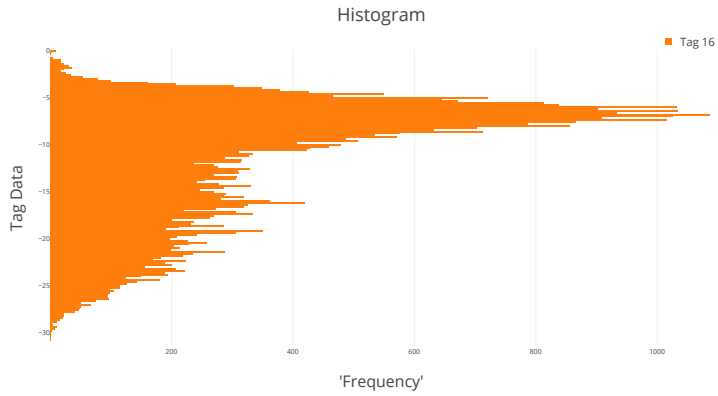


(c)

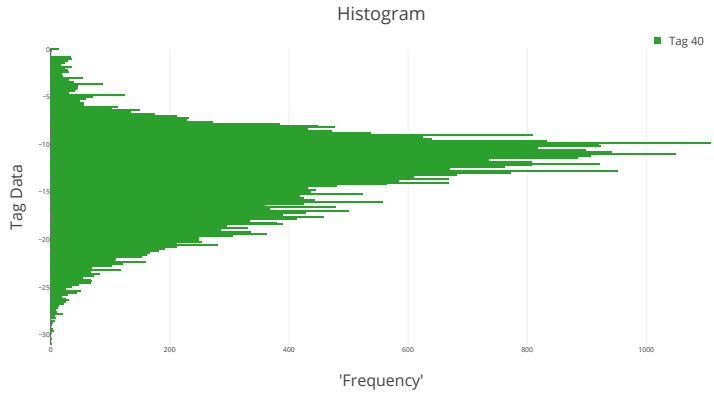
Figure 6.2: Histogram plots with two histograms overlaid. **(a)** shows the SNR value of all tag detections from 6th of November until 10th of may i both cages. **(b)** shows the same, but only for tag messages between 7 in the morning and 16 in the afternoon. **(c)** shows the same, but only for tag messages between 16 in the afternoon until 7 the next morning.



(a)

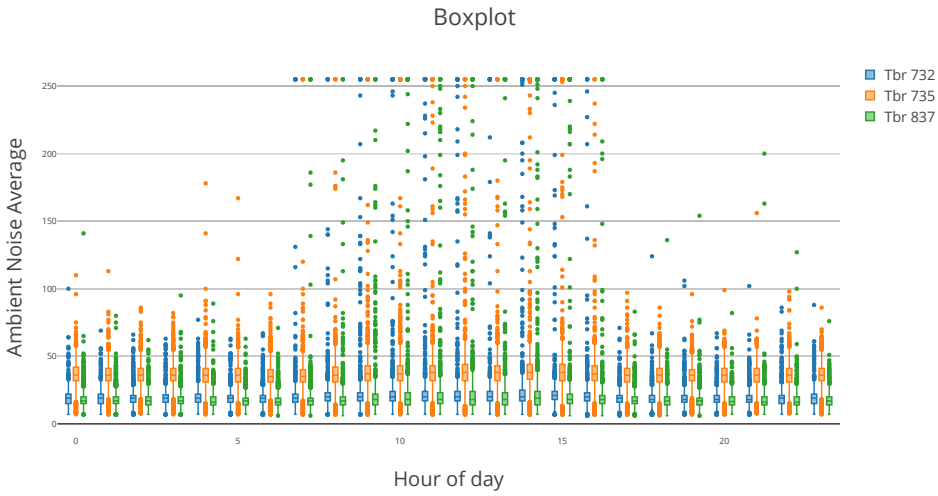


(b)

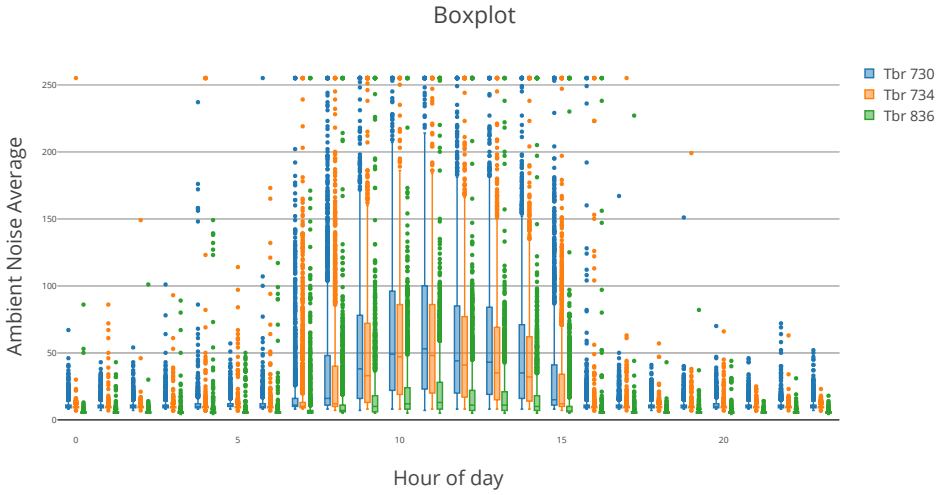


(c)

Figure 6.3: Horizontal histogram plots of depth data for three tags in Aqautraz.

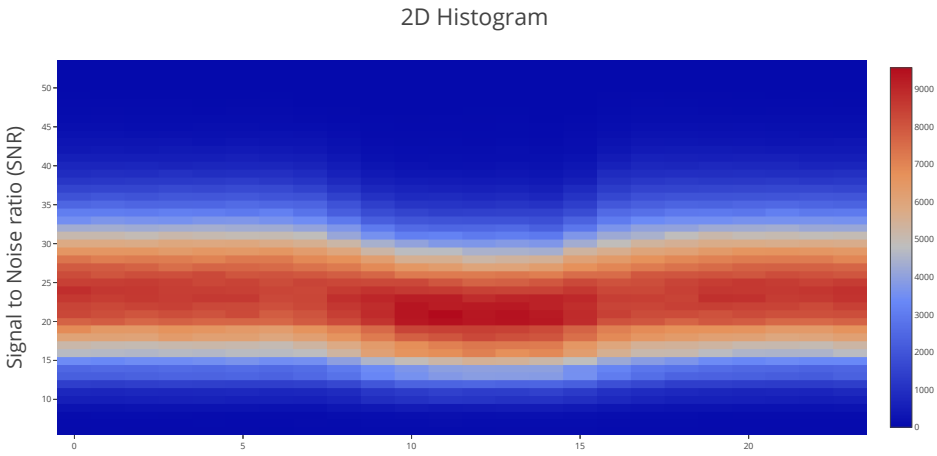


(a)

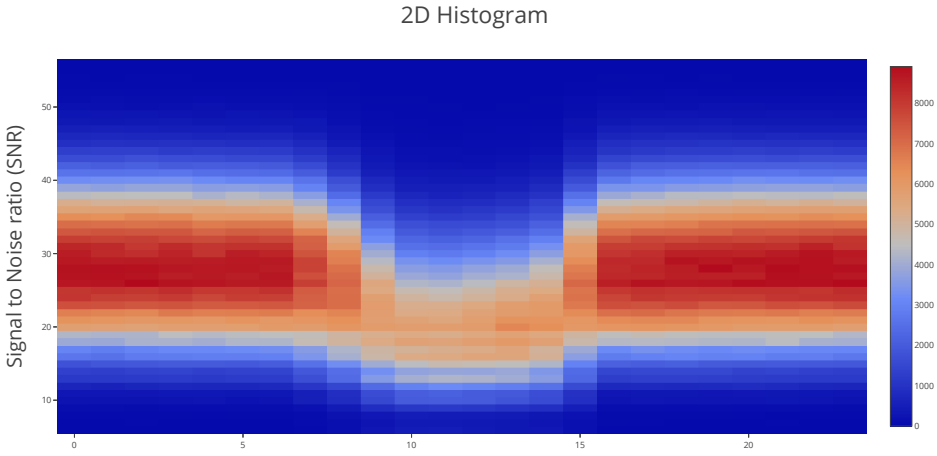


(b)

Figure 6.4: Boxplots with hour of day on the x-axis and average ambient noise value on the y-axis. **(a)** Boxplots for the Aquatraz cage **(b)** Boxplots for the reference cage



(a)



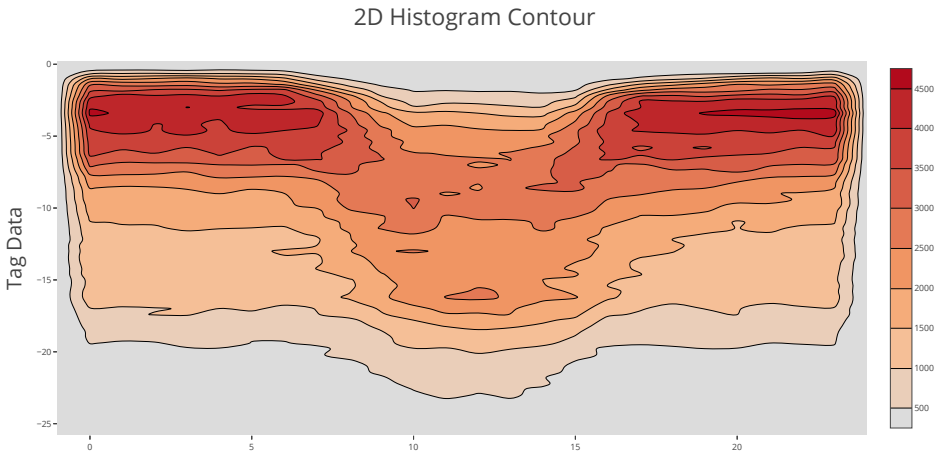
(b)

Figure 6.5: 2D histogram of SNR and time of day. Each column in the graphs is one hour of the day. **(a)** 2D histogram for Aquatraz cage **(b)** 2D histogram for Reference cage

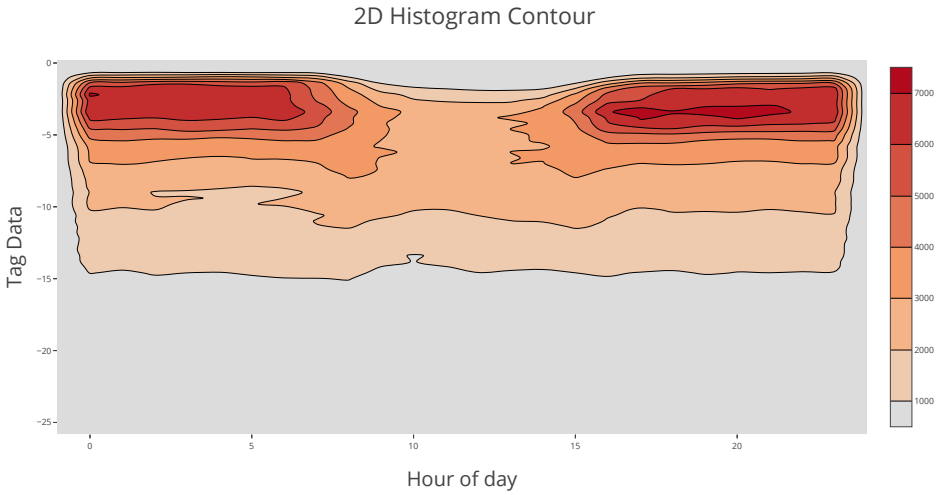
Figure 6.6 shows examples of 2D Histogram contour plots in Dash, in this case 2D histogram contour plots of time of day and depth in both case study cages. These plots show a similar trend to fig. 6.5, where the values midday are different from the rest of the day. Again, the reference cage plot can be seen to have less intensity during midday as well. For these plots, depths below 25 in Aquatraz were ignored in an effort to maintain equal axes between the two graphs. The tag messages with a lower depth than 25 did not affect the contour plot of Aquatraz, so no valuable information was lost due to this filtering. Figure 6.12 in the next section shows a similar plot, but for ambient noise average over the project period, and fig. 6.13 shows a time of day depth contour plot of all detections with the lowest possible SNR value, 6.

Figure 6.7 shows an example of a 3D scatter plot in Dash, in this case a 3D position plot of found positions of two tags (10 and 97) in the Aquatraz cage. This plot type is at the time of writing not possible to export from Dash or plotly in vector graphics format. The cage model is accurate in and also its relation to the location of the Aquatraz TBRs. The cage model was plotted as three circles (line plots), and several line plots originating in that circle, leading down to the lowest circle, before meeting in the bottom center point of the cage. The middle circle in the plot, best seen in figs. 6.7c and 6.8, is located at 8 meters depth, indicating that the windows in the steel frame of Aqauatraz were located below this circle. The triangle shows optimal locations in terms of PDOP (section 2.2.4). A similar cage model without the middle circle was implemented for the reference cage. Tag 10 was rarely found at a depth above 5 meters or below 25 meters. Tag 97 on the other hand was evenly distributed around the whole cage. The position fixes were filtered both by using time of arrival, and distance from center of cage. The distance threshold was set to 1.1× cage radius. As seen in the figure, this allowed some locations to fall outside the cage.

Figure 6.8 shows an example of a 3D scatter animation plot in Dash, in this case showing one frame of a 3D position animation for one tag. This plot type is at the time of writing not possible to export from Dash or plotly in vector graphics format. The animation plot allowed the user to play and pause as they pleased, and while paused, the user could rotate, pan, zoom etc. as they desired. The rangeslider below could also be dragged to manually adjust the current frame. The animation was limited to 15 points per frame

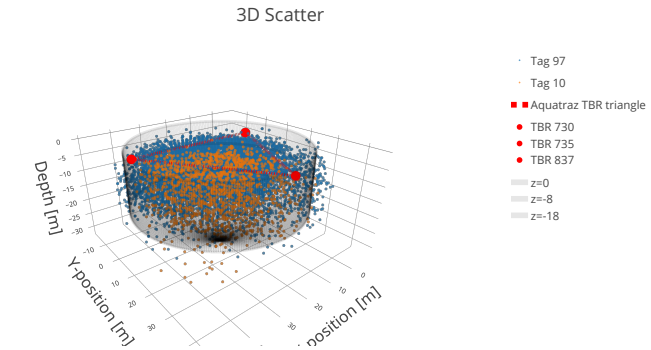


(a)

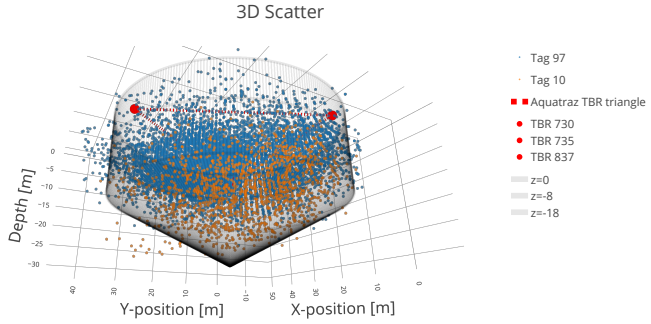


(b)

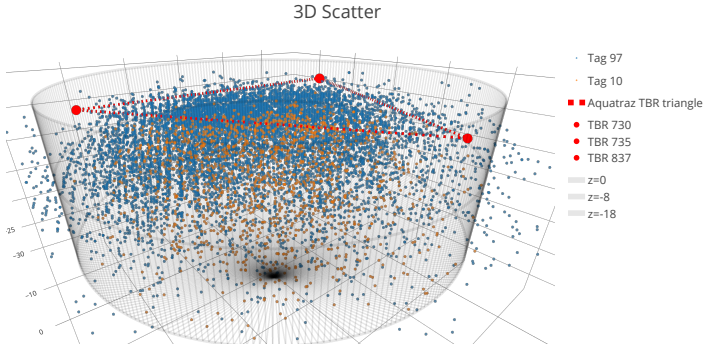
Figure 6.6: 2D contour histogram plots of depth and time of day. **(a)** 2D contour histogram for Aquatraz cage. There was data for depth 25 to 32 meters too, but they were not included for this plot. Does not make a visual difference, so equal axis between the two subfigures was prioritized. **(b)** 2D contour histogram for Reference cage



(a)



(b)



(c)

Figure 6.7: 3D scatter plot of two tags. **(a)** shows a view far-off. **(b)** shows the cage from down under. **(c)** show a zoomed-in view from the same angle as **(a)**.

to better follow the movement of most recent position, highlighted in yellow. The user would be warned with a pop-up to avoid selecting large date periods, as this would increase processing time greatly. The steps of the rangeslider are nonlinear due to the nature of positioning in this system.

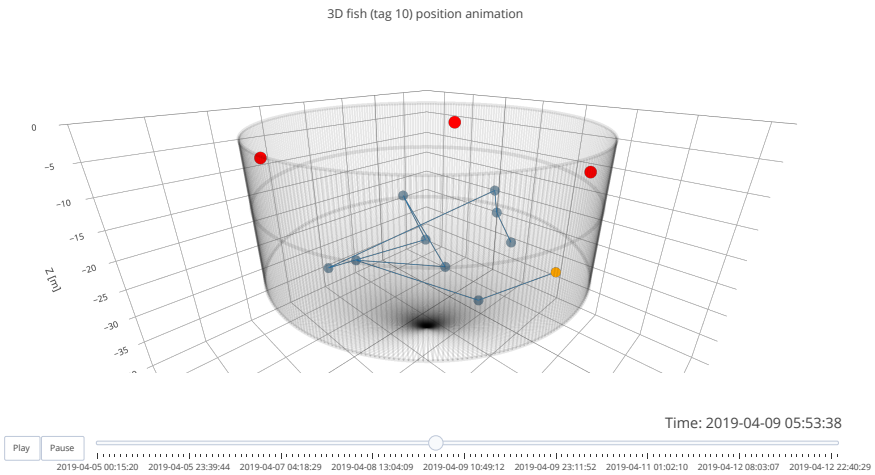


Figure 6.8: A 3D animation was also implemented in the app. Buttons allowed the user to play and pause the animation. One could also pause the animation at any time, zoom around, rotate, and also drag the animation slider manually.

6.4 Additional case study figures

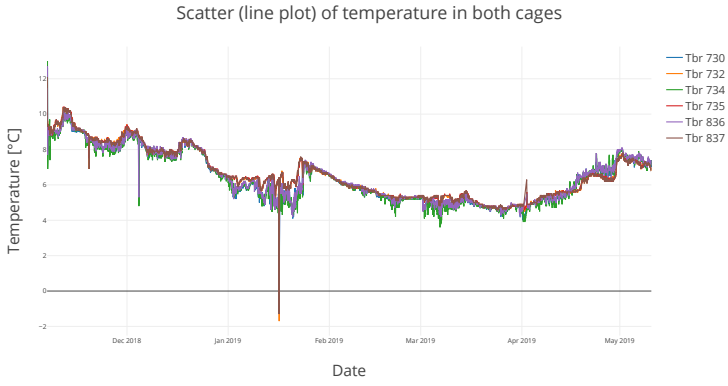
Additional graphs building on the figures presented in section 6.3 is presented in this section, showcasing insightful data from the case study. These figures, figs. 6.9 to 6.23, are included to fully demonstrate the versatility a tool like the IoF app can offer. A brief discussion on the main results of the data in the figures of this chapter is given in chapter 7.

Figure 6.9 shows how temperature developed through the case study period, from 6th of November 2018 to 10th of May 2019. By comparing figs. 6.9b and 6.9c, one can see that the temperature was more stable in the Aquatraz cage than the reference cage, where the difference in temperature between each TBR was greater. The spikes of the graphs are from removing the TBRs from the water due to changing batteries and offloading data from them. When a TBR is lowered back into the water, the internal temperature sensor needs to readjust to the water temperature, thus creating spikes on plots like these.

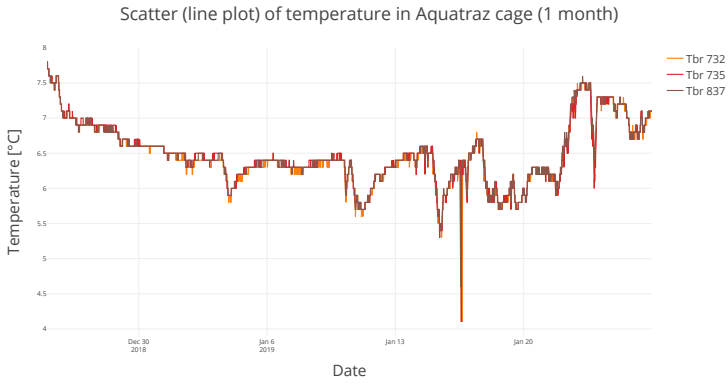
Figures 6.10 and 6.11 reveal zoomed-in views of fig. 6.4. In fig. 6.10 the boxplots go from midnight until 5 in the morning. Again, one can see more clearly that one TBR in Aquatraz had a consistently higher noise average value. This is the TBR located to the "north-east" in fig. 5.3. The boxplots go from 10 in the morning until 15 in the afternoon in fig. 6.11. The noise average is unchanged for the Aquatraz cage, but the observed noise average in reference has increased greatly for two of the TBRs, with slightly higher values in the third. The boxplot with lower values was the one located "south" (downwards) in fig. 5.3.

Figure 6.12 shows how the noise average has developed over the project period in both cages. As one can see in fig. 6.12a, the noise average Aqautraz increased in November before it slowly decreased until early February, where it stabilized, and then further decreased somewhat after passing into March. The average noise value was also varying less as time passed. In contrast, fig. 6.5b shows an unchanging noise average in reference. The average noise was quite low compared to the Aquatraz cage throughout, becoming even more concentrated and stable in March.

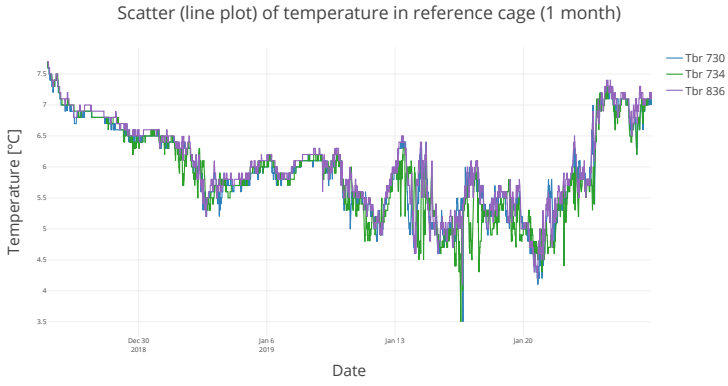
Figure 6.13, like fig. 6.5, shows a contour plot of hour of day and depth of tags. However, in this figure, only tag detections with the lowest SNR value possible is included. So the data use are all observed tag detections in both cages with SNR 6. As both plots



(a)

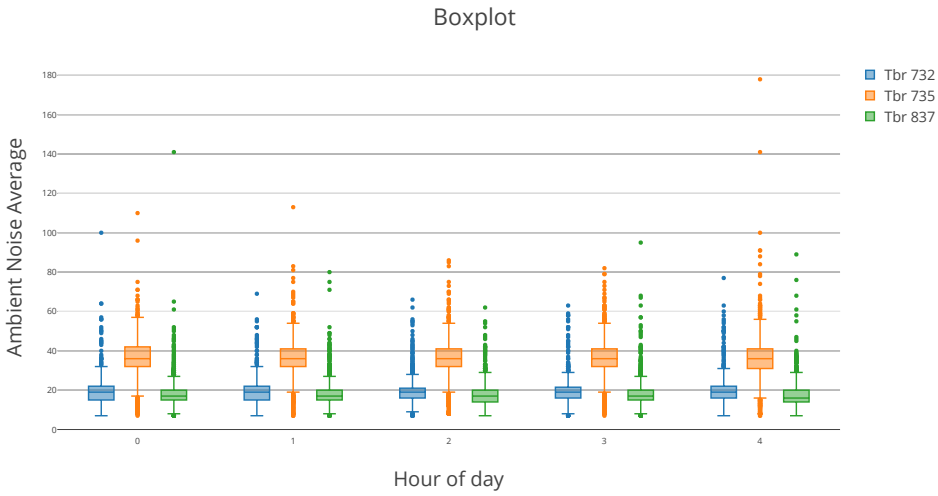


(b)

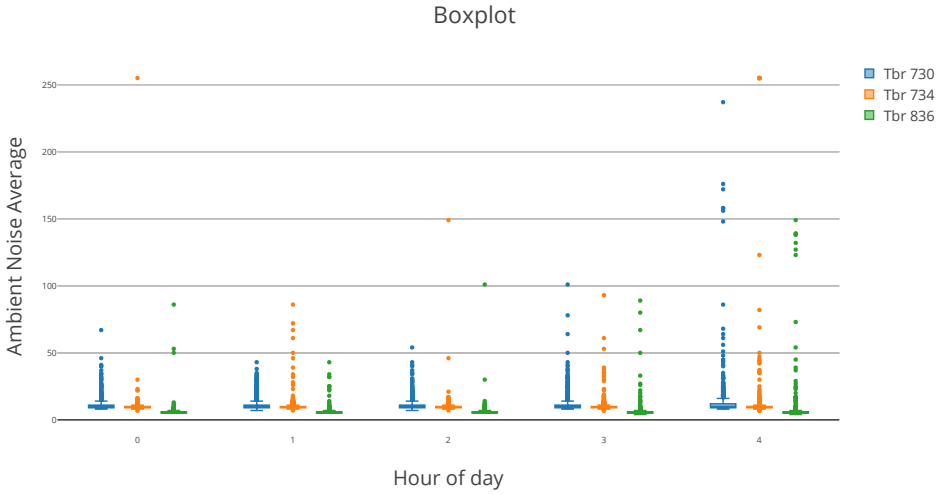


(c)

Figure 6.9: Line plots of temperature in both cages. The spikes in the plots are from removing the TBRs from the water to offload local data and change batteries. **(a)** Shows temperature readings for all TBRs from 06.11.2018 - 10.05.2019. **(b)** shows temperature in Aquatraz 25.12.2018-25.01.2019. **(c)** shows temperature in reference 25.12.2018-25.01.2019.

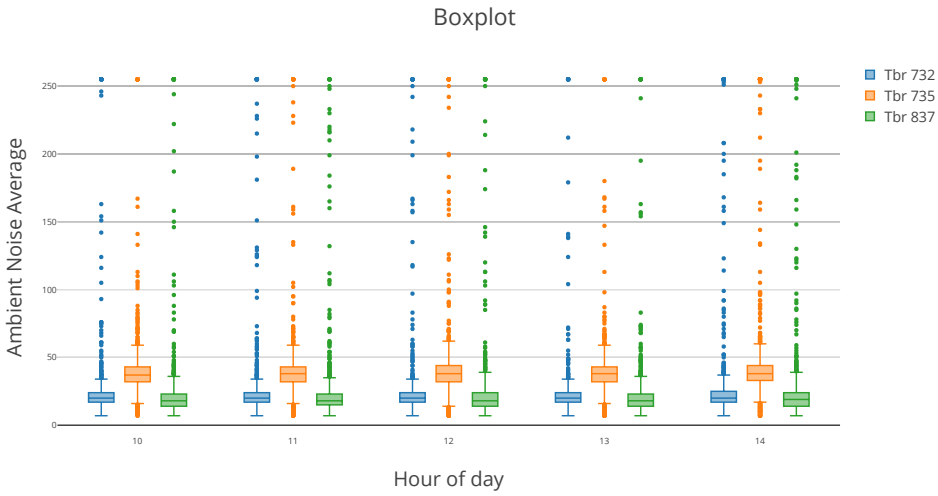


(a)

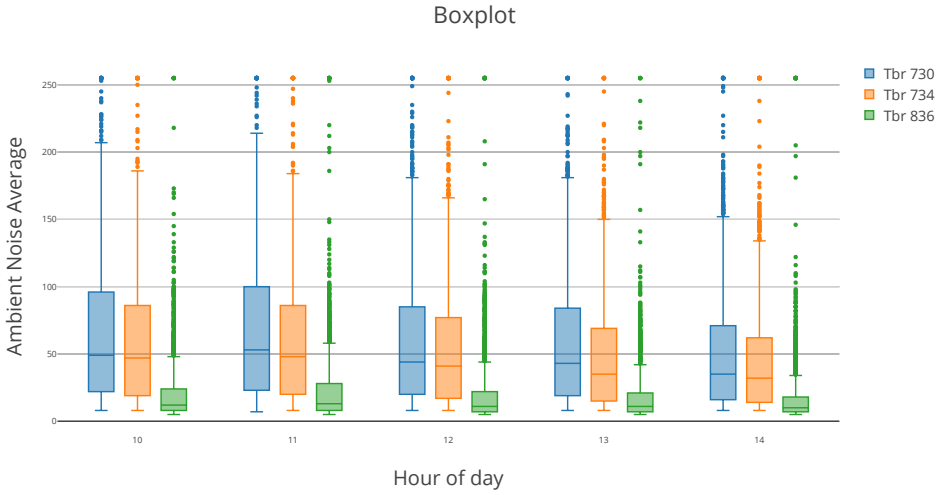


(b)

Figure 6.10: Boxplots with hour of day on the x-axis and average ambient noise value on the y-axis. **(a)** Boxplots for the Aquatraz cage **(b)** Boxplots for the reference cage

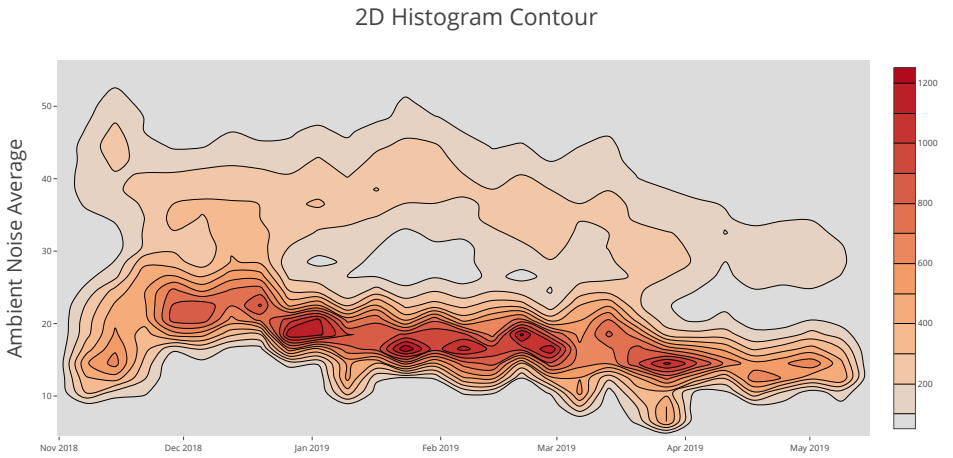


(a)

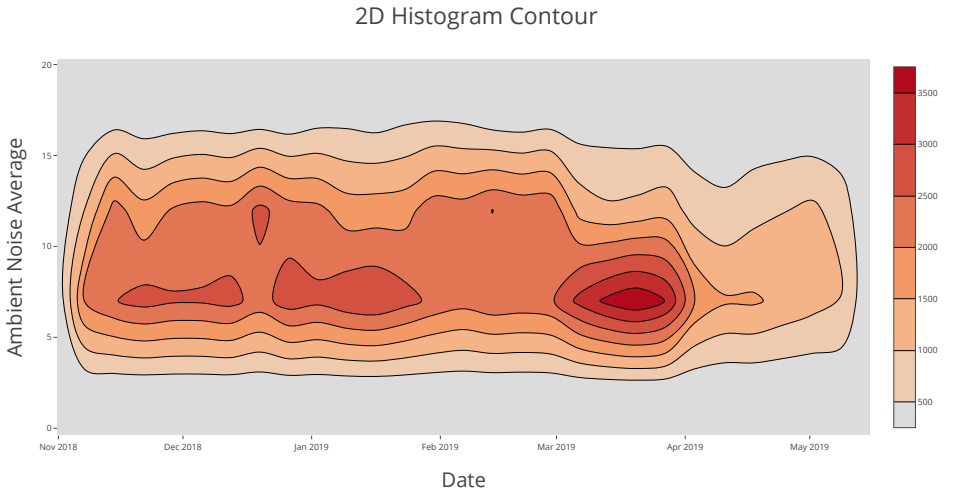


(b)

Figure 6.11: Boxplots with hour of day on the x-axis and average ambient noise value on the y-axis. **(a)** Boxplots for the Aquatraz cage **(b)** Boxplots for the reference cage



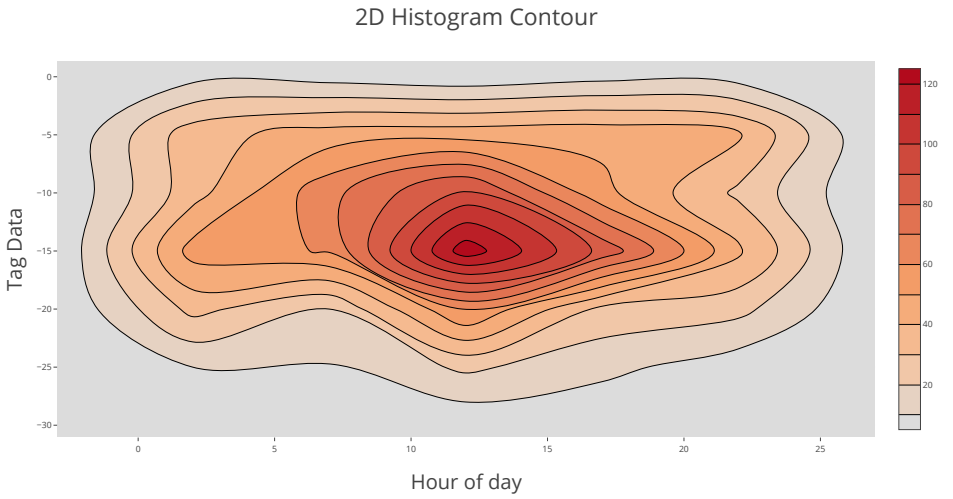
(a)



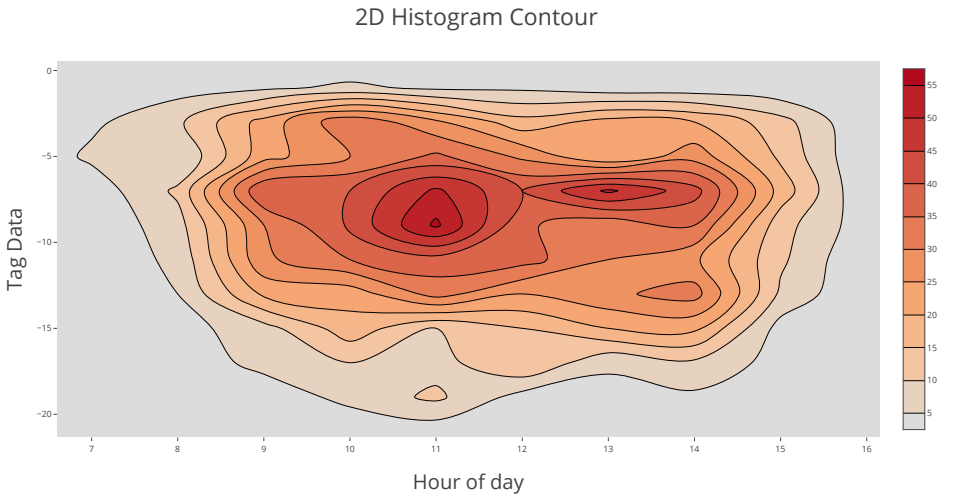
(b)

Figure 6.12: 2D contour histogram plots of ambient noise average and date. **(a)** 2D contour histogram for Aquatraz cage. **(b)** 2D contour histogram for Reference cage.

show, these signals are typically occurring around midday. In addition, far less tag detection with this poor SNR was observed in reference, where less than 5 were observed before 7 and after 16 on any given day.



(a)



(b)

Figure 6.13: 2D contour histogram plots of depth and time of day of all tag detections. Data visualized are only from tag detections with the lowest possible SNR value, 6. **(a)** 2D contour histogram for Aquatraz cage. **(b)** 2D contour histogram for Reference cage. Notice that for the reference cage the x-axis only goes from 7 in the morning to 16 in the afternoon

Figure 6.14 is a line plot based on data received through the IoF back-end after 10th of May. It is included here to showcase how the gateway lost connectivity several times, before going completely dark 10th of June.

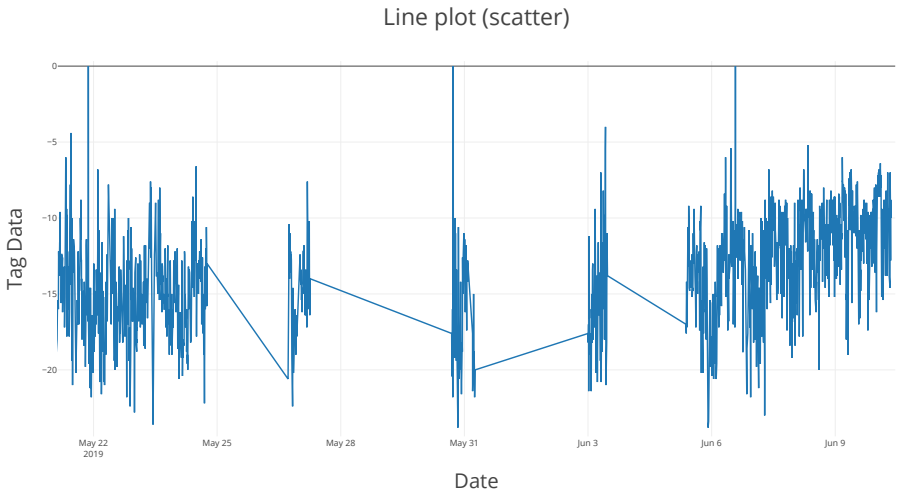
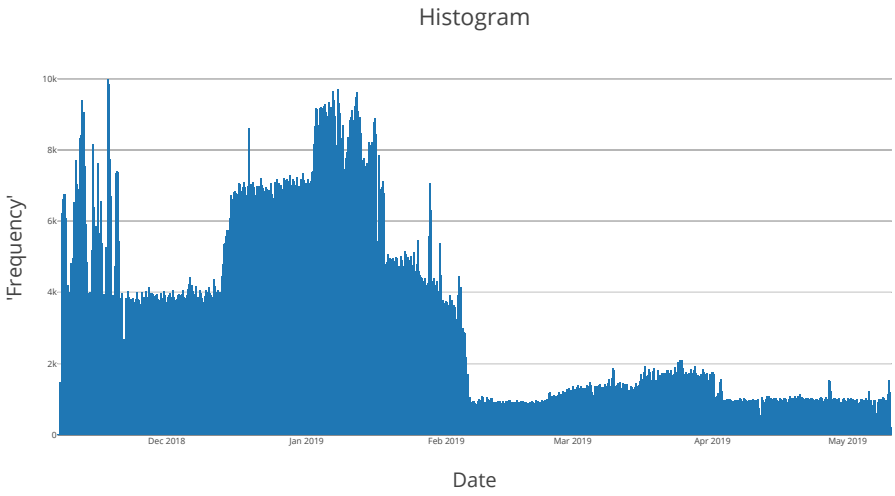


Figure 6.14: Line plot showing periods of gateway offline after reinstalling gateway 10th of may through a timeseries depth plot for a tag. The gaps in data are due to gateway going offline. The gateway did not come online again after June 9th.

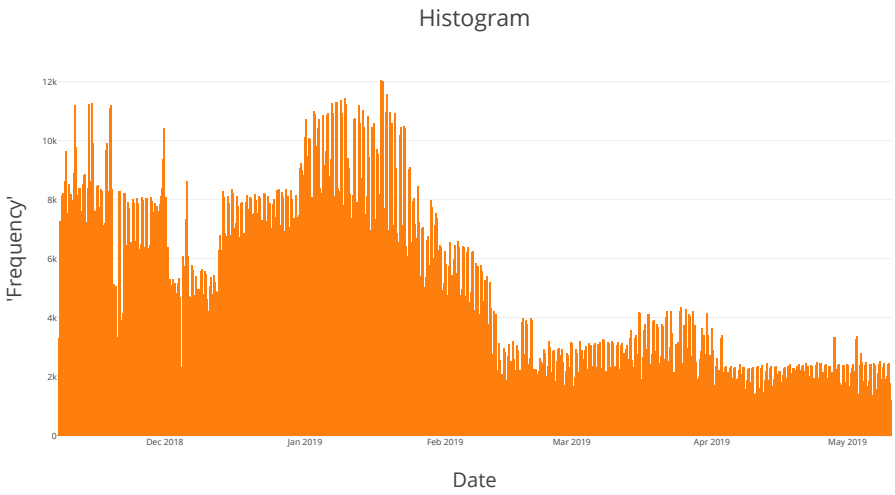
Figure 6.15 consists of two histograms that shows number of detections throughout the case study, where each column represent number of detections 6 hours of a day. The first column starts at 9 in the morning 6th of November. An oscillating pattern was observed in the reference cage detections (fig. 6.15b), that gradually became less pronounced as time went on. Other than that, both cages showed similar number of tag detections in a similar trend, although the reference cage had less sharp drops and rises, and had higher peaks than Aqautraz.

Figure 6.16 shows number of tag signals from Aqautraz successfully detected in the reference cage. In other words, this figure shows the leakage of data from Aquatraz to reference that was present. While leakage did occur the other way around, not nearly as many detections were found.

Figure 6.17 shows the noise average distribution in both cages, where fig. 6.17a shows the distribution from 06.11.2018 - 07.02.2019, and fig. 6.17b shows the distribution from



(a)



(b)

Figure 6.15: Histogram of all tag detections in both cages. Each bar represents all tag detections within 6 hours, starting at 9 in the morning for the first bar 6th of November. number of detections that day. **(a)** Histogram detections in Aquatraz cage. **(b)** Histogram detections in reference cage.

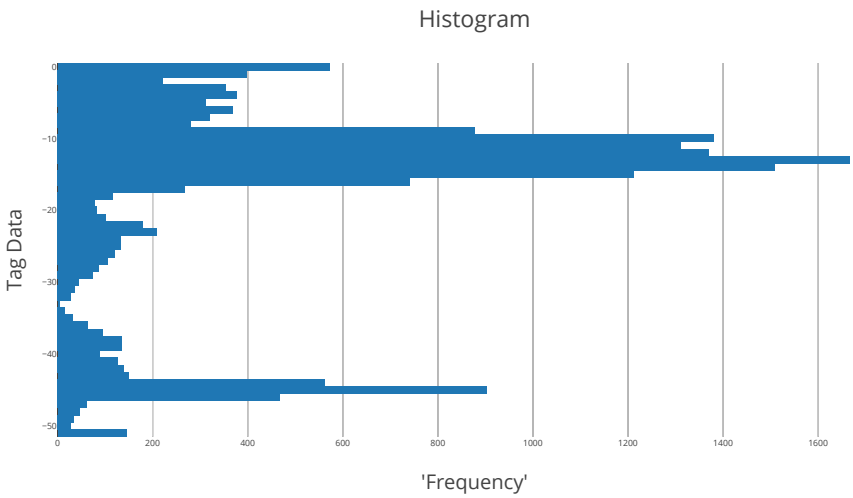


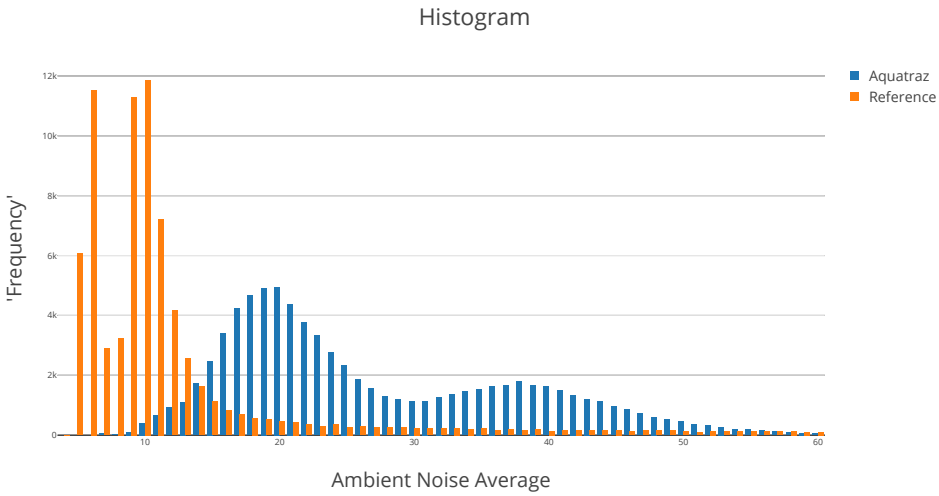
Figure 6.16: Horizontal histogram of Aquatraz leakage to reference cage. This plot shows tag signals with origin in Aquatraz that have been successfully received in the reference TBRs. The bars represent number of times a signal has been detected at that particular depth.

07.02.2019 - 10.05.2019. Reference was observed, like in fig. 6.12, to be more or less constant, while Aquatraz became less prone to high noise, though still higher than reference.

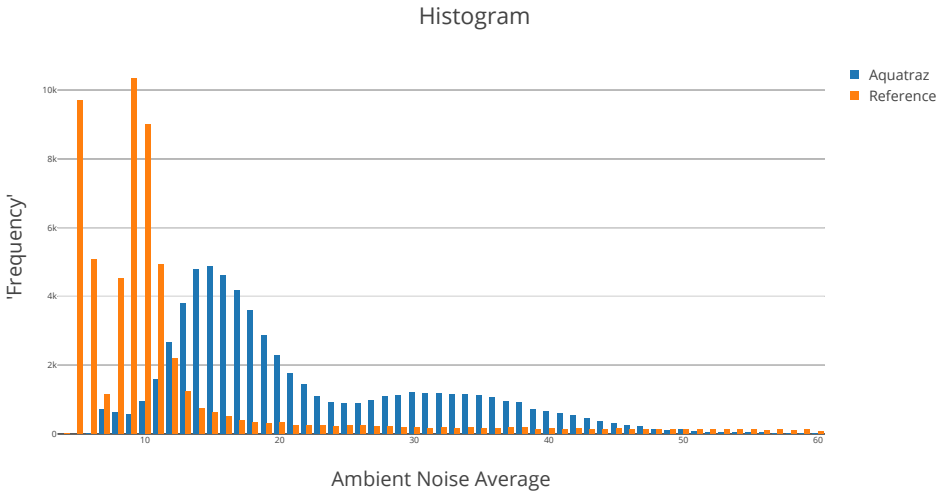
Figure 6.18 shows how SNR values developed for tags with frequency 69 throughout the project period. Both figures, i.e. both cages, was observed to have a decrease in SNR from December until mid-February, where they stabilized.

Figure 6.19 show number of successful TDOA positions found with one day for each column. For Aquatraz (fig. 6.19a), a sharp drop happened 18th of December, before regaining positioning 16th of January. 10th of February they positions died out again, before more solutions started to be found 02. April. For the reference cage (fig. 6.19b), a similar story, except nothing between 18th of December and 2th of January, except a few in early March.

Figures 6.20 and 6.21 shows the xy contour plots of tags that had frequency 71 and 73, i.e. tags that in this case study sent only depth data, but did so around every 40 seconds, from start 06.11.2018 until around 7th of February 2019. Figure 6.20g is an example of a tag that there was found many positions for. Figure 6.21i is an example of a tag that there was not found many positions for. The xy-coordinates are defined as outlined in section 2.2,

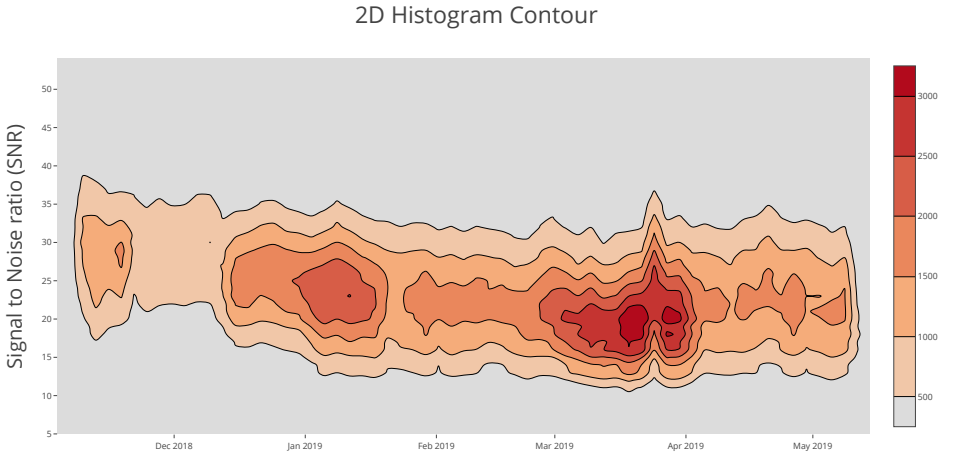


(a)

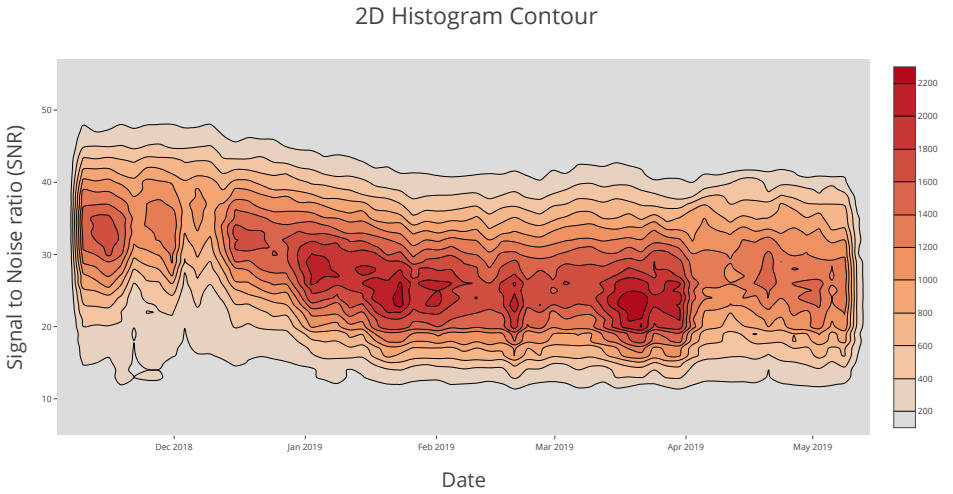


(b)

Figure 6.17: Histogram showing number of times different noise averages have been detected. Noise average values over 60 is not included in the graph, as there were few occurrences compared to the the rest. **(a)** Shows noise average histogram from 6th of November until 7th of February **(b)** Shows noise average histogram from 7th of February until 10th of May.

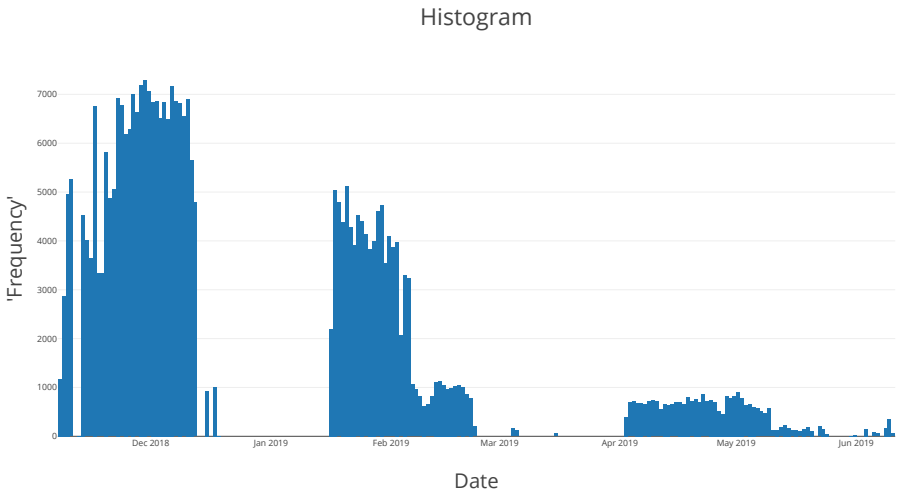


(a)

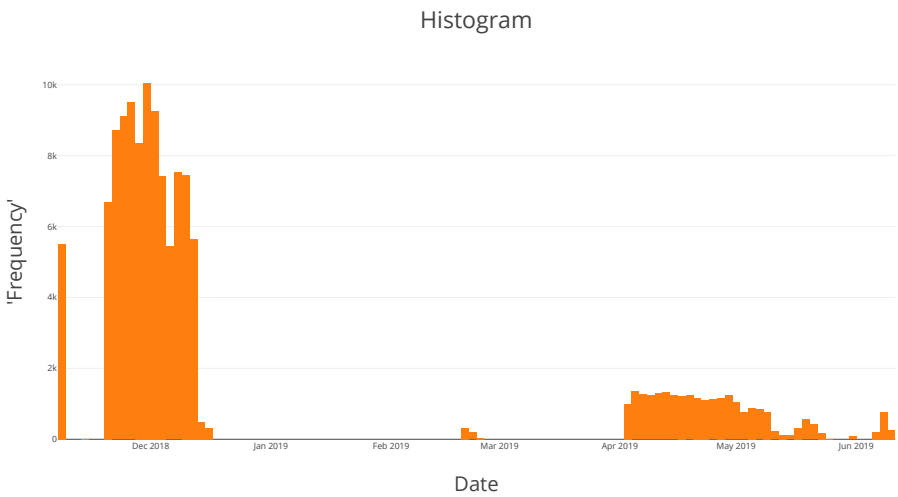


(b)

Figure 6.18: 2D histogram contour that shows SNR compared to dates of tags with frequency 69. **(a)** SNR development Aquatraz cage. **(b)** SNR development reference cage.



(a)



(b)

Figure 6.19: Histogram of date of found positions in both cages. Each bar represent 2 days, the height being number of positions found in those 2 days. **(a)** Shows found positions in Aquatraz **(b)** Shows found positions in reference

with TBR A, B, and C following the setup in fig. 5.3. As fig. 6.19 shows, no positions were found in reference between 18th of December 2018 and 2. April 2019. This means that there was far fewer positions underlying the reference cage plot in fig. 6.21. Figures 6.22 and 6.23 shows the same, but this time for all tags with frequency 69, which sent depth around every 2 minutes.

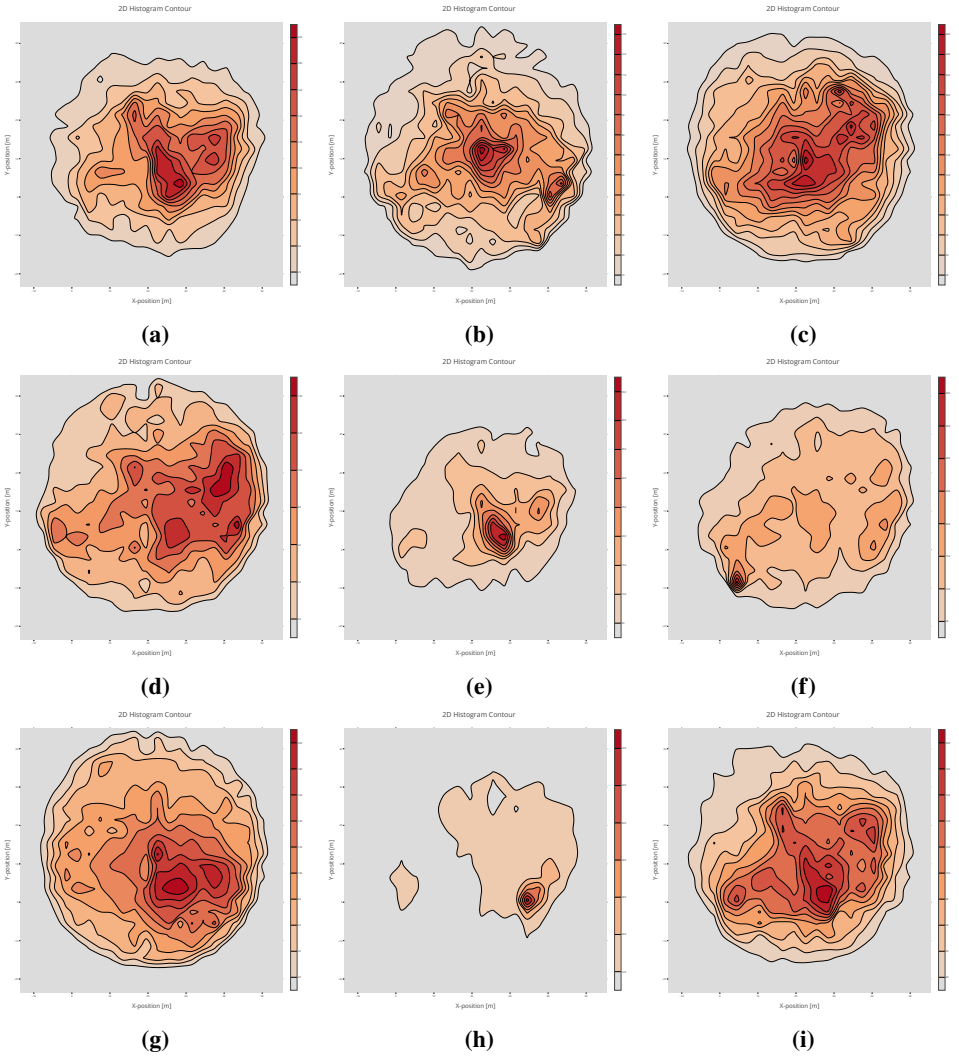


Figure 6.20: XY contour plot of tags with frequency 71 and 73 in Aquatraz.

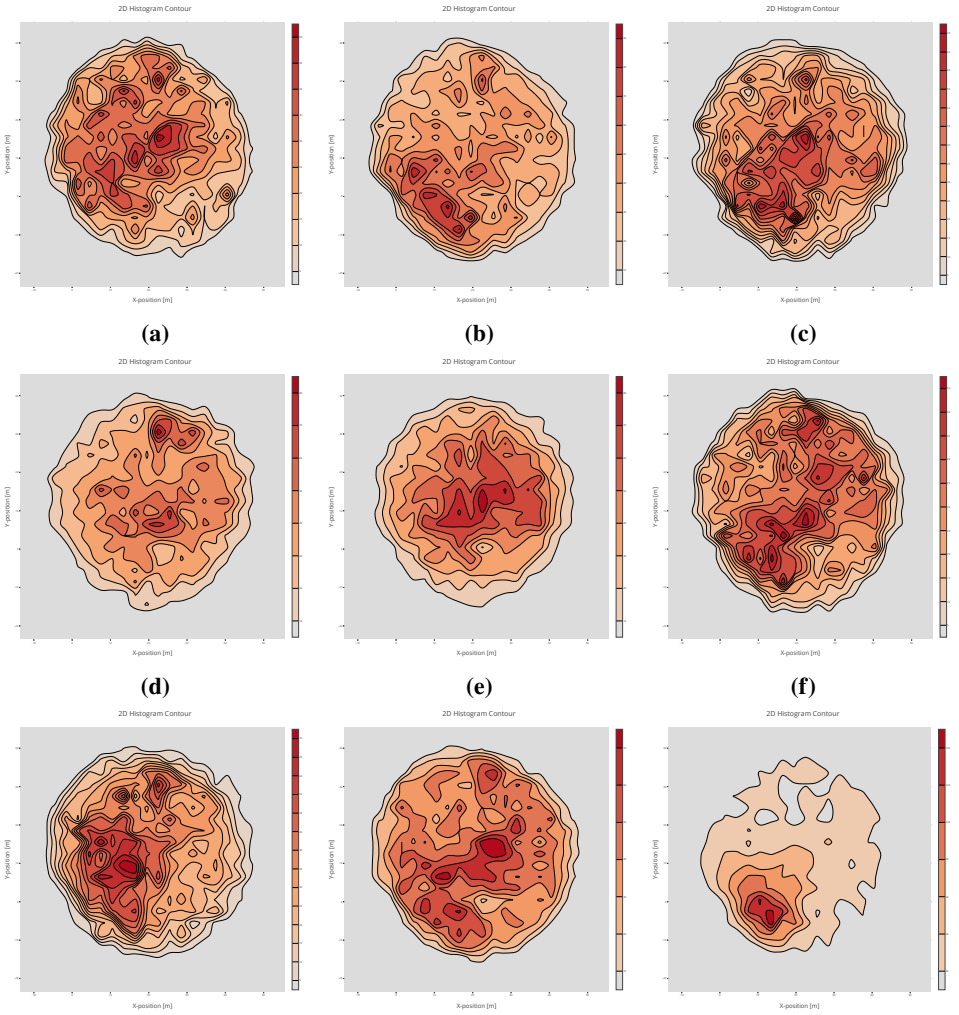


Figure 6.21: XY contour plot of tags with frequency 71 and 73 in reference.

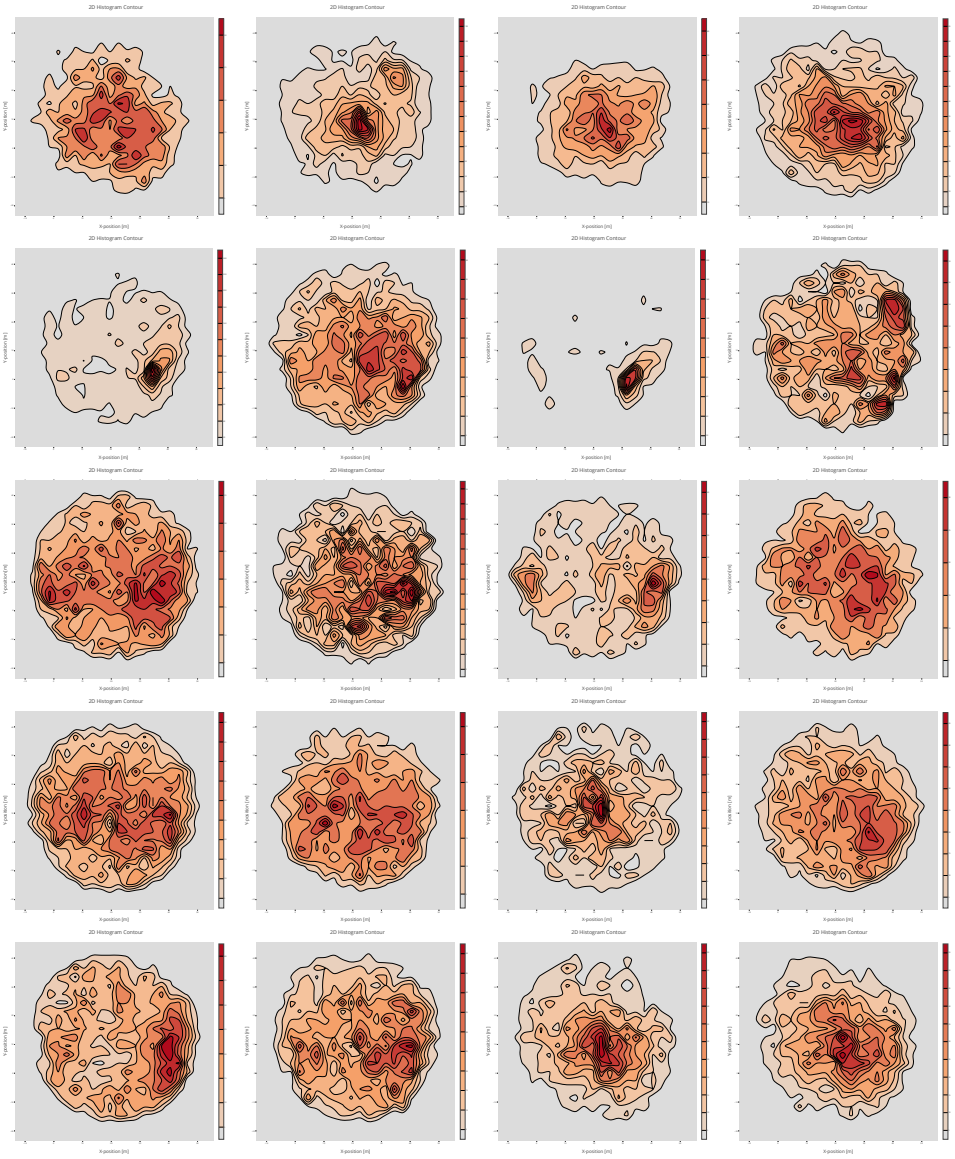


Figure 6.22: XY contour plot of tags with frequency 69 in Aquatraz

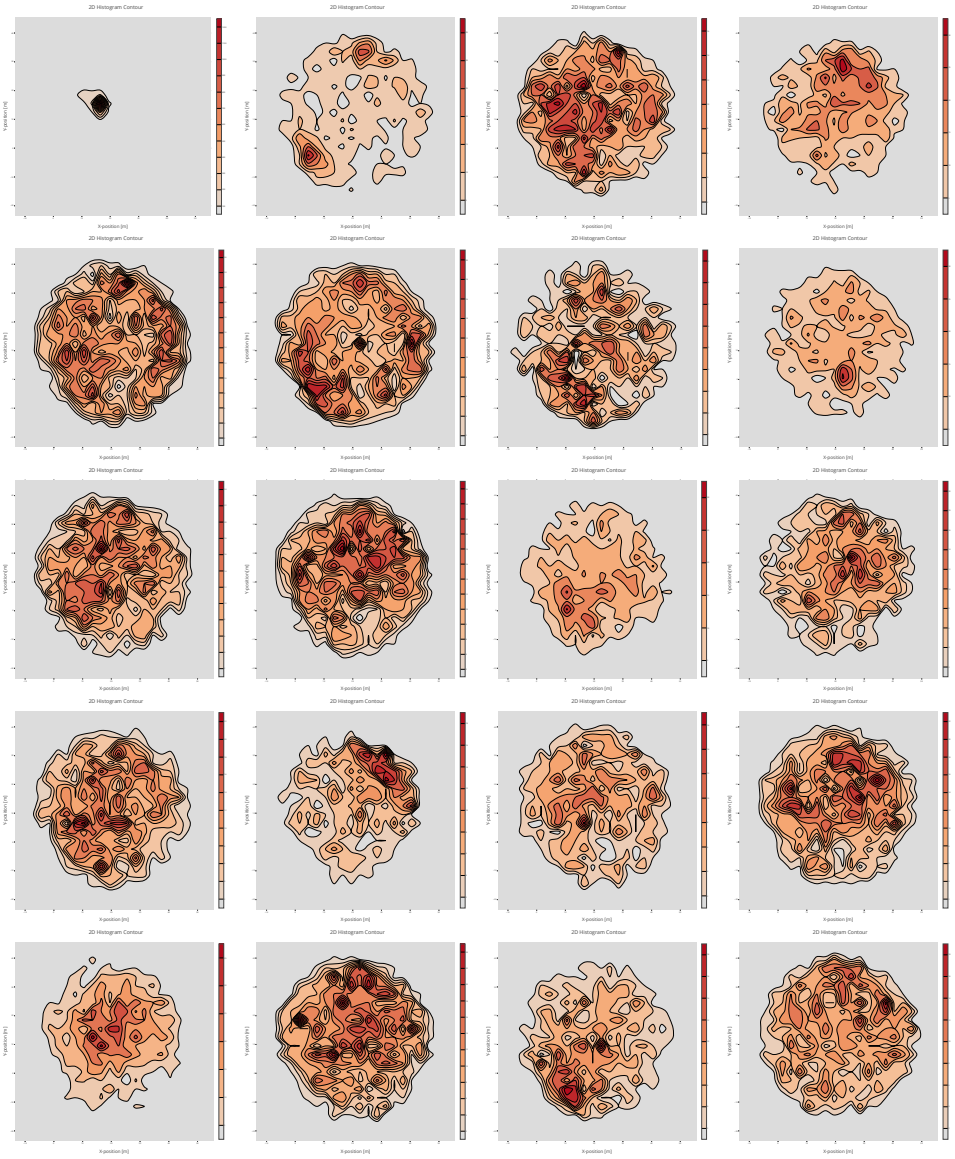


Figure 6.23: XY contour plot of tags with frequency 69 in Aquatraz

Chapter 7

Discussion

Details on the design and chosen implementation of the IoF system was given in chapter 4, with accompanying discussion for the various parts. In this chapter, the main results of the development of IoF is discussed through the lens of the project goals presented in chapter 1. Improvements for future work on the IoF system is also presented.

7.1 The IoF concept

Acoustic telemetry has been deemed feasible in commercial sea cages to monitor depth of fish in real-time by [11], where acoustic communication in the sea-cages were not strongly impaired by factors such as fish density and local noise. While the results of data transmission in the reference cage are not as favorable in this use case, the performance overall is quite good, and it provides valuable data not possible to obtain without acoustic telemetry. Acoustic telemetry has long proved its usefulness in research on wild fish, and it has proven its place in commercial sea cages too. The sound environment is important, and an unforeseen consequence of instrumenting fish for the case study was tag leakage. A potential factor to reduce corruption of data is a stronger checksum in the acoustic transmissions.

When viewing the data from the case study from the original back-end application, it was clear that the capacity of LoRa was breached several times. This problem has been

alleviated with a new protocol that is able to send many more messages per minute, most likely more than the TBRs are able to detect within each transmission. The configuration of LoRa can also be modified to allow more transmission, but this will be at the cost of spread factor and range. Alternatively, one could modify the SLIM to transmit data in periods of lower tag interference, to better optimize the bandwidth usage, but this somewhat defeats the purpose of using acoustic telemetry to monitor fish in real-time. LoRa is generally not well-suited for real-time application due to the limitations on data rate, but in an acoustic tag setting, the bottleneck of data rate will often be in the acoustic messages. The battery performance using LoRa is great, and so the networking protocol is well-suited for IoT-devices in conditions where battery is important. In comparison, using mobile network solutions from telecommunication providers drains battery life greatly. Considering the issue of reaching data limits solved, one can be comfortable with LoRa as a solution for the IoF system.

7.2 Aims of the thesis

The first goal was to identify and describe the requirements to the IoF system. Having no knowledge about acoustic telemetry, LoRaWAN, and MQTT at project start, Fall semester 2018, this was an important step. A NTNU specialization course on oceanographic instrumentation and biotelemtry [23] was of great help in this regard, and by self-studying, the concept of MQTT and LoRaWAN was also approached. Hands-on experience with hardware from Thelma BioTel was gained through the case study. With a SLIM module and TBR already designed to function as the perception layer, and a chosen gateway for the network layer, the requirements of these two layers were easy to define. The overall network architecture was also quite clear, but some refinements to better explain the flow of data from a tag in a fish to a hard drive far away was done during the thesis development. The requirements of the application layer is however another story.

As an cybernetics engineering student there was no background or experience to lean on when it came to web technology and web development. In addition, having close to no experience with Python, or large application software development alone, a lot of topics needed to be delved into. A new world of best practices, programming language

specific features, and principles and tools needed to be understood. Spending time to research what options exists to fill the vacant spot of the application layer, finding out their differences, experimenting with implementing simple versions of them, and learning concepts such as SQL and the MVC model. While the back-end software developed during this period worked with the then current perception and network layer, the message handling was poorly developed, and it needed thorough rework when implementing new message formats. While very valuable experiences were gained during this period, the goal and purpose of the application layer was less clear, exemplified in loosely define requirements to the application layer. Better defining the goals of the application layer was therefore an early priority for the thesis work. With a backpack full of knowledge and insight in Python programming for web software applications, a more thorough approach to defining requirements for application software was completed. This bore fruits, as can be seen in the detailed requirements of chapter 3. It is by these requirements the IoF system is evaluated.

The next main objective of the thesis was to identify all relevant data types and define corresponding message formats, protocols, and network transports from end-device to IoF back-end. Related to this goal was the goal of implementing the full IoF protocol on the SLIM hardware, preferably with better compression methods to optimize LoRaWAN bandwidth. As already mentioned, the main groundwork for the new message protocol was done in the specialization project, in the form of suggestions of improvements. The data types of the system was also identified. Revisiting those suggestions, and iterating on them further, the current protocol format was reached. This was presented in detail in chapter 4. However, no time so far in this paper has been spent on development for the SLIM hardware.

As the main focus of this thesis is the application layer, little focus has been given to the SLIM module, despite around a month of work to understand, design, and implement the full working protocol in SLIM. A large portion of that time was spent on debugging unknown errors, like incompatible SDK versions, as is typical for hardware development. The software for SLIM is written in `C`, and is developed by co-supervisor Waseem Hassan. Understanding a relatively large codebase without any prior knowledge takes some time. Luckily, the code was written so that the main modifications needed were only in two

functions dealing with unpacking data received from TBRs. Bit and byte manipulation was written here to achieve the new protocol format. In the LoRa packing of messages, modifications were also needed, to accommodate the new header format. And SLIM packet sending needed to be implemented.

The SLIM software currently supports all possible communication protocols, SLIM IoF packets, and it uses less bytes. Before, the maximum number of detections was 11, which would use 123 bytes (with header). During the case study in the fall, 10 tags sending a message every 40 seconds was present in both cages. Thus the data load limitations could be fully tested. However, this could not be tested in the same manner after implementing the new protocol, since the batteries of these tags had died out. Despite this, some IoF messages containing 17 detections were received after updating SLIM modules at Aquatraz site, most likely due to some buffer buildup in SLIM resulting in two IoF messages as one. 17 detections + an accompanying SLIM packet and header use 118 bytes. It is clear that this is a major gain in data rates, allowing for more complex instrumentation of experiments. The implementation in SLIM was in other words a success. There are still problems with the SLIM, however, such as the timestamp drifting, and at times inconsistent GPS readings.

Choosing and implementing a flexible solution for storage of IoF data was also an important objective. The different options and thoughts behind them are presented in detail in chapter 4. The end-result is an `SQLite` database perfectly suited for the IoF system. But there are some additional considerations that is worth mentioning as a design flaw of the currently implemented solution. First of all, `PRIMARY KEY` can be defined uniquely by using multiple columns to do so in the IoF database tables. This is a discovery made late after completing development, which is why it was not implemented. It would improve database lookup time. Another important factor is the fact that post-processing of IoF data takes a lot longer than reading from database, converting timestamp to datetime to string being the most time consuming culprit. By preprocessing this transformation and adding it as an additional column in the database tables would increase performance greatly, potentially to the point where reading all from a database file can be sufficiently fast.

Finally, the objectives of the study was to demonstrate IoF by implementing a IoF

web-based application front-end for the case study, targeting 2/3D visualization of fish space usage in Aquaculture sea cages. And to plan and conduct tests to validate the IoF concept with the described use case. The validation of the IoF concept was done through the requirements to the application layer, and the exported plots from the front-end are demonstrations that illustrate the IoF concept for the use case. In addition to the material presented in chapter 6, a short video series was made demonstrating the the application live [21]. Looking through the requirements, one can see that most of the functional requirements are met. At the same time, most of the non-functional and UI requirements are not met. These would greatly increase the quality of the IoF app as a tool for future use. The application performed well for the case study, but it is important to criticize the software quality of this part of the system.

First of all, the code remains largely undocumented. While typical dash code is quite self-explanatory, proper documentation is always useful. Second, there is a lot of code duplication that most likely could be avoided. While some steps were taken to reduce code duplication, with the chosen design for the webpage layout, it cannot be avoided. Third, the application is one big Python file. Dividing the different parts logically elsewhere would improve quality of code, for example all callbacks in a separate file. Furthermore, the application code will only work for the Aquatraz case study. While it is true that the design and meat and logic of the application can be reused elsewhere, the code is so intertwined with the project that it would be hard for an outsider to know where to begin pulling it apart. Lastly, work should be done to look into how to better solve the problem of sharing data between callbacks. While the in-memory object store works well, perhaps it is a solution with more overhead than needed for most IoF experiments. The focus was mostly on getting everything to work, not necessarily the best way to do so. Thus, the quality of this component is worse than the rest of the system. Luckily, these flaws cannot be experienced by the end-user in the web browser.

Regarding quality of back-end, it is well-formed. More or less all requirements for back-end have been meet, including NFR and UI requirements. Improving documentation, enabling project-specific MQTT topic subscription, and enabling logging options through a command line interface, remain as NFR and UI requirements not met. These are minor

parts that can easily be improved upon. Improving the persistent storage by including timestamp converted to date as a string, and using `PRIMARY KEY` constraint also remains as useful steps to take for improving the overall IoF system. Also, publishing back-end as a package on the Python Package Index (PyPi) could also be useful to make installation even easier, as it would enable installation through the `pip` manager.

7.3 Case-study data analysis

There are some interesting results found from the case study that is worth writing briefly about. The findings concern both fish behaviour and consequences of the chosen case design. However, no work has been done to remove biases from the data, or to process the data in any additionally capacity. As such, the following analysis is not meant to give definitive answers to the situation of the cages, but rather illustrate some of the interesting and unknown data that can be observed with the IoF system.

First of all, most of the fish seemed to follow a pattern similar to the one shown in fig. 6.1. Figure 6.1a shows how two fish alternate between depths, in a near perfect opposite pattern. Similar behavioural patterns were found in [11], where the fish swam deeper in a time frame of the same length, centered around midnight and not during daytime like tag 46 of the figure. The same behaviour with deeper depths during the daytime, and closer to the surface, was observed in [13]. Furthermore, fig. 6.3 shows how the fish seems to have preferred depths that vary between them. Individual behaviour is complex, and there are many factors that can influence what is seen through the IoF data, such as temperature, feeding, genetics, and environmental parameters like current and salinity.

Figure 6.6 shows where the fish stay during the different hours of the day. Interestingly, at least in the Aquatraz cage, the fish seems to follow the pattern of staying close to the surface, and then swimming deeper during the active window. In the reference cage, the amount of detections go down significantly within this window. This can be seen in fig. 6.15b, where all spikes have rapid increase and decreases. Notice also that the size of these variations decrease mid February. This coincides well with the tags sending updates frequently dying out late January / early February, which is also reflected in the number of detections in the figure. Some degradation of quality in this window can also be seen in

Aquatraz when viewing fig. 6.5a. But the total detections seem not to be affected like they are in reference.

This is further seen when viewing figs. 6.4, 6.10 and 6.11. It is clear how the noise is much more varied in reference with much higher values in the active time period. The interesting part is that the two TBRs closes to the Aqautraz cage are the two reporting much larger noise average values. Figure 6.16 shows that Aquatraz tag signals leaking to the reference cage definitely happen, and while it happens from reference to Aquatraz too, it is by far a lot less. The fish swimming deeper during the active window means that they swim around the windows, giving clear line of sight to the reference cage. This could explain the increase of noise average during these hours, in addition to a more complex image if one assumes that the fish swims around more actively. The Aquatraz TBRs are protected by the cage the other way around. In Aquatraz, the one furthest away from the reference cage have a consistently higher value than the others. Using line of sight as argument, this could make sense, since it will be the most exposed to the reference cage. When viewing boxplots for the limited data after 10th of may, when the setup of TBRs in Aqautraz was moved (fig. 5.2), TBR 732 becomes the one with the absolute worse noise, and TBR 837 becomes pretty bad to. Actually they follow the pattern of reference, low good values, except for the active window.

This can indicate that leakage is not the sole reason for degradation in noise quality. The case after 10th of May is actually a good one. The TBRs are hanging 10 meters deep, right outside Aquatraz windows, getting clear line of sight to most of Aquatraz and reference all the time. Despite this, the noise average becomes around 5 times worse from 7 until 16 for TBR 732. Why not constantly worse? 735 actually improved greatly, becoming very stable and low in noise, except some more noise midday. And why is the effect less pronounced in the other two, especially 735 which actually got better for all times of the day. It is difficult to tell for sure as underwater acoustics is incredibly complex. One would believe that the noise average situation would improve over time as more tags go offline, but apparently it does not, at least in the reference cage figs. 6.12b and 6.17.

For Aquatraz on the other hand, the noise average did improve over time, seemingly following a downwards trend in tact with the frequent updating tags coming offline (fig. 6.12a).

This fact makes sense. The noise average measurement in TBR is a quite narrow band that is set to listen at 69 KHz in the case study TBRs. So it is in reality only viewing the ambient noise of that specific frequency. In reference, normal water, the tag signals are distinct from each other. But inside Aquatraz, reflections in the wall will lead to distortions of signals, as a sort of filter, leading to some of the tags with frequency 71 or 73 being detected within the ambient noise bandwith of the TBRs.

Figure 6.18 shows a weird result. The SNR of tags with frequency 69 becomes worse from December to February, in both cages, although the data is more spread in the Aquatraz case. And then a peak in the end of March in Aqautraz with improved SNR. This is most likely related to the total number of detections in frequency 69. Figure 6.15 shows all detections, including 71 and 73, but when viewing the detections for only 69, it is observed that when the detection number goes up quite a lot, the SNR value at the same time goes down, which makes sense. The sudden peaks up and down in number of detections is difficult to explain, as the acoustic factors are complex. But it is clear that the number of detections stabilize early February after the last 71/73 tags die out.

Figure 6.19 shows when the batteries of the SLIM modules died. The sharp drop of positions found stem from TBR clocks going out of sync, a clear sign of SLIM battery dying. The sharp comebacks are from directly after replacing batteries. It is unclear why barely no positions were found from 16th of January to 2nd of April in the reference cage. Figures 6.20 to 6.23 shows XY contour plots of the fish in both cages. It is clear that some fish prefer to stay at certain parts of the cage, assuming the data can be trusted. One can also see that some of the fish have fallen out, resulting in a densely concentrated contour plot from a static location.

There are two reasons to present this analysis. One, to illustrate some of the limitations of the case study. To properly compare how fish behave in an Aquatraz cage compared to a reference cage, it would be beneficial to remove the underlying causes for noise and degradation, and to switch batteries in SLIM modules before they die, thus finding more positions to potentially gain a deeper understanding of their swimming behaviour. It is also unfortunate that the acceleration data was mostly ruined. A solution for the noise problem could perhaps have been to instrument the fish in Aquatraz with frequencies 68, 70, and 72.

That way, tags interfering with each other would be less of a problem. However, another reason why the noise and complexity increase from 7 in the morning to 16 in the afternoon could be because of day-to-day operations. Equipment such as boats driving around the site can influence the acoustic conditions. If that is the case, it would be difficult to eliminate, and the case study would accurately reflect how the Aquatraz cage differs.

The other purpose of including this analysis is to showcase the breadth of questions that arises from having access to visualization of data like the IoF app offers. The main focus was on noise and limitations of the study, but countless hours have been spent just browsing through the data, looking for clues, seeing connections and so on. That is the best demo to illustrate the value of the IoF concept. While the analysis presented here might not find the correct reasons for why we see what we see, it does not matter that much, as the focus of this paper is not on the actual data (beyond benefits and flaw of the IoF system), but rather the IoF concept as a whole. And one thing is for sure, it definitely inspires the mind to take a deep dive into the data and find answers. And that is what IoF is all about, learning by exploring data. Gaining a peek into the lives of fish that no other data gathering method can offer.

Chapter 8

Concluding remarks

8.1 Summary

In this thesis, the Internet of Fish concept has been presented and realized. The system uses acoustic telemetry, local wide area network, and internet to transmit data from fish to the internet. In the perception layer of the system, the acoustic tags produce signals that real-time hydrophones receive, which synchronization and LoRa Interface modules transmit with LoRa to a gateway in the Network layer. This gateway subsequently converts the data to MQTT and sends it to a MQTT server in the application layer, which forwards it to the appropriate subscribers. Here a back-end MQTT client is running to receive the data, unpack it, and store it persistently in storage, and perform difference of time of arrival positioning on the incoming data. Finally, front-end software polls the same database for new messages, and makes the data available to end-users through a front-end web application.

The concept was proven and tested with a use case study. Previous work in the specialization project used the same case study, but did not support an interactive web application, and both back-end software and message compression was flawed. For the case study, acoustic tags were placed in 30 fish in the Aquatraz sea cage, and 30 fish in a normal sea cage on the same site location, owned and operated by Midt Norsk Havbruk AS. The tags

and hydrophones used are produced by Thelma BioTel, and the synchronization and LoRa Interface module is made by PhD-candidate Waseem Hassan and associate professor Jo Arve Alfredsen. The software for the module was updated during development to support a new design of the IoF message protocol that compress data. Sending status packets with positional data was also implemented. The gateway used is the MultiConnect© Conduit ™produced by MultiTech. A virtual `Ubuntu 18.04 LTS server` was set-up on NTNU to run the MQTT server. The back-end client application developed for this thesis was also run on this virtual server, and the local development computer. The message handling of back-end was completely redone, and major other parts of the back-end was also rewritten, refactored, and expanded. TDOA-based positioning was implemented in back-end. A full web application that showcase the potential of the IoF concept was made with Dash by Plotly, and data collected from the use case was presented through the front-end. The front-end application was specifically made for the case study, but making it work for other projects in the future seems feasible.

The project work has shown that the Internet of Fish concept is a viable concept for real-time monitoring of live fish in commercial sea cage situations. The data from the use case seem to suggest varying individual behaviours of the fish, as well as differences in behaviour between the cages. Some differences are to be expected since Aquatraz is a larger and deeper cage made of steel. In the Aquatraz cage, the noise conditions are generally worse than in the reference cage, but the hydrophones are less affected by worsened ambient noise conditions from 7 in the morning until 16 in the afternoon. A new IoF message protocol was implemented, greatly saving on the amount of data being transmitted. The data collected has been explored, but not in great depth, as it was only meant to illustrate the power of IoF in the context of this thesis. Gateway has disconnected multiple times when testing the system, before losing connection completely. All other components of the IoF system has been running without downtime. No unwanted behaviour has surfaced in back-end or front-end of the application layer. The implementation of IoF is now fully implemented, making IoF a true reality. The culmination of the past year of work has resulted in a useful tool to have for both commercial implementations and research.

8.2 Future work

There are a couple of improvements for the different layers of the IoF system that should be explored in the future.

Firstly, finding ways to further optimize battery usage in SLIM modules would be beneficial, to maintain synchronized times longer. One way this could be achieved would be to have a more precise clock in TBR that does not drift out of sync as quickly. That would limit how often SLIM needs to synchronize time. Implementing battery status in the SLIM packet, and potentially other status information, would be a natural next step to take when developing SLIM further.

Secondly, finding a better gateway solution. The gateway has proved to be a major problem. Power outages happen, and for an unknown reason, gateway is not able to reconnect to MQTT broker after an power outage, and it also seem to randomly disconnect and connect (6.14), possible due to bad internet connection. Switching over to a battery and mobile network based solution would probably improve uptime, making real-time monitoring of fish more of a reality.

Thirdly, some suggestion for back-end was suggested in section 7.2. These are relatively minor, and should be easy to implement. Further improvements to do besides these are to better accommodate wild fish experiments. While the code is interchangeable, positioning especially should be revisited so that it can handle both wild fish and cage scenarios. This too should not prove too much of a challenge.

Finally, for the front-end, several things can be done. They have mostly been mentioned already, but to reiterate: Add documentation, make the application project agnostic, metadata file handling, explore other options for sharing data between callbacks, add features such as proper figure export, data table view, data export, data upload, dashboard functionality and more. The application possibilities are many. And since the whole system is modular, each component can easily be swapped for others. Another gateway can be used, some other form of hydrophone and synchronization module can be used (although this would probably include redesign of IoF protocol), different front-end, and even different back-end. Perhaps some experiments are better suited with a different front-end design than the one used. Having the option to easily switch between layouts and themes in quick

configurations would be a nice feature to have.

Regarding the case study, a more comprehensive analysis of the data should be done. This analysis should look into factors responsible for data loss, and to eliminate potential factors affecting data integrity. Parameters unavailable at the time of writing, such as day-to-day operations (feeding, crowding etc.), would also be beneficial to include in the analysis and front-end application.

Bibliography

[1] L. Alliance. What is the lorawan™ specification? https://lora-alliance.org/about-lorawan, 2018. [Online; accessed 05-December-2018].

[2] A. Arrow. The plasma in-memory object store. https://arrow.apache.org/docs/python/plasma.html. [Online; accessed 26-May-2019].

[3] M. AS. http://www.multitech.net/developer/products/multiconnect-conduit-platform/conduit/. [Online; accessed 05-Deceomber-2018].

[4] A. Banks and R. Gupta. Mqtt version 3.1.1. plus errata 01. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html, 2015. [Online; accessed 05-December-2018].

[5] D. Berckmans. Automatic on-line monitoring of animals by precision livestock farming. *Livestock Production and Society*, 01 2006.

[6] T. biotel as. http://www.thelmabiotel.com/. [Online; accessed 05-December-2018].

[7] D. by Plotly. Sharing data between callbacks. https://dash.plot.ly/sharing-data-between-callbacks. [Online; accessed 13-May-2019].

[8] G. Dudek and M. Jenkin. *Computational Principles of Mobile Robotics*. Cambridge University Press, New York, NY, USA, 2000.

[9] J. I. Efteland. Underwater acoustic positioning system for real-time fish tracking. Masterś thesis, NTNU, 2016.

[10] B. T. Fang. Simple solutions for hyperbolic and related position fixes. *IEEE Transactions on Aerospace and Electronic Systems*, 26(5):748–753, Sep. 1990.

[11] M. Føre, K. Frank, T. Dempster, J. Alfredsen, and E. Høy. Biomonitoring using tagged sentinel fish and acoustic telemetry in commercial salmon aquaculture: A feasibility study. *Aquacultural Engineering*, 78:163 – 172, 2017.

[12] M. Føre, K. Frank, T. Norton, E. Svendsen, J. A. Alfredsen, T. Dempster, H. Eguiraun, W. Watson, A. Stahl, L. M. Sunde, C. Schellewald, K. R. Skøien, M. O. Alver, and D. Berckmans. Precision fish farming: A new framework to improve production in aquaculture. *Biosystems Engineering*, 173:176 – 193, 2018. Advances in the Engineering of Sensor-based Monitoring and Management Systems for Precision Livestock Farming.

[13] M. Føre, E. Svendsen, J. Alfredsen, I. Uglem, N. Bloecher, H. Sveier, L. Sunde, and K. Frank. Using acoustic telemetry to monitor the effects of crowding and delousing procedures on farmed atlantic salmon (salmo salar). *Aquaculture*, 495:757 – 765, 2018.

[14] Google. Google python style guide. `https://google.github.io/styleguide/pyguide.html`.

[15] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15:287–317, 1983.

[16] W. Hassan, M. Føre, J. B. Ulvund, and J. A. Alfredsen. Internet of fish: Integration of acoustic telemetry with lpwan for efficient real-time monitoring of fish in marine farms. *Computers and Electronics in Agriculture*, 163:104850, 2019.

[17] W. Hassan, M. Føre, H. A. Urke, T. Kristensen, J. B. Ulvund, and J. A. Alfredsen. System for Real-time Positioning and Monitoring of Fish in Commercial Marine Farms Based on Acoustic Telemetry and Internet of Fish (IoF). *International Society of Offshore and Polar Engineers.*, June 2019. *(Accepted, 29th International Society of Offshore and Polar Engineers (ISOPE) conference)*.

[18] T. Haugen, T. Kristensen, T. Nilsen, and H. Urke. Vandringsmønsteret til laksesmolt i vossovassdraget med vekt på detaljert kartlegging av åtferd i innsjøsystema og effektar av miljøtilhøve, 06 2017.

[19] P. A. Kjelsvik. Internet of fish. Engineering cybernetics, specialization project (ttk4550), NTNU, 2018.

[20] P. A. Kjelsvik. Iof. `https://github.com/perkjelsvik/iof`, 2019.

[21] P. A. Kjelsvik. Youtube playlist demonstrating iof front-end case-study web application. `https://www.youtube.com/watch?v=rpEMRQNQsTo&list=PLQUpHVL89YjZB15o1xn9gbc6551HjL54V`, 2019. [Online; accessed 24-June-2019].

[22] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, Aug. 1988.

[23] kreklev. Ttk15 oseanografisk instrumentering og biotelemetri. `https://www.itk.ntnu.no/emner/fordypning/ttk15`, 2015. [Online; accessed 01-July-2019].

[24] S. S. Løvskar. Positioning of periodic acoustic emitters using an omnidirectional hydrophone on an auv platform. Masterś thesis, NTNU, 2017.

[25] R. Minerva, A. Biru, and D. Rotondi. Towards a definition of the internet of things (iot). `https://iot.ieee.org/definition.html`, 2015. [Online; accessed 05-December-2018].

[26] D. Narayanan and O. Hodson. Whole-system persistence with non-volatile memories. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. ACM, March 2012.

[27] M. Newton, J. Barry, J. Dodd, M. Lucas, P. Boylan, and C. Adams. Does size matter? a test of size-specific mortality in atlantic salmon salmo salar smolts tagged with acoustic transmitters. *Journal of Fish Biology*, 89, 06 2016.

[28] M. norsk Havbruk AS. https://mnh.no/aquatraz/. [Online; accessed 06-December-2018].

[29] Oasis. Mqtt version 5.0. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, March 2019.

[30] J. V. Plas. Python's visualization landscape. https://speakerdeck.com/jakevdp/pythons-visualization-landscape-pycon-2017, May 2018.

[31] Plotly. Comparing webgl vs svg in python. https://plot.ly/python/compare-webgl-svg/. [Online; accessed 04-April-2019].

[32] plotly. Bioinformatics, now with the power of dash. https://dash.bio/, 2019. [Online; accessed 13-April-2019].

[33] Plotly. Welcoming dash 1.0.0. https://medium.com/@plotlygraphs/welcoming-dash-1-0-0-f3af4b84bae, June 2019.

[34] T. Preston-Werner. Tom's obvious, minimal language. https://github.com/toml-lang/toml.

[35] Python. Changelog. https://plot.ly/python/compare-webgl-svg/. [Online; accessed 30-June-2019].

[36] Sphinx. Sphinx - python documentation generator. hhttp://www.sphinx-doc.org/en/master/.

[37] SQLite. Datatypes in sqlite version 3. https://www.sqlite.org/datatype3.html. [Online; accessed 18-April-2019].

[38] G. stackexchange user whuber. https://gis.stackexchange.com/questions/8650/measuring-accuracy-of-latitude-and-longitude/8674#8674. [Online; accessed 20-June-2019].

[39] P. Trefethen, U. Fish, and W. Service. *Sonic Equipment for Tracking Individual Fish*. Special scientific Report. U.S. Department of the Interior, Fish and Wildlife Service, 1956.

[40] u-blox. *u-blox 8 / u-blox M8, Receiver Description Including Protocol Specification*, May 2019.

[41] H. Urke, T. Haugen, G. Kjærstad, J. A. Alfredsen, and T. Kristensen. Laks- og aurebestanden i strynevassdraget; vandringsmønsteret hjå laksesmolt og aure, ungfiskproduksjon og botndyr. - mina fagrapport 48. 56 s., 03 2018.

[42] G. van Rossum. Pep 8 – style guide for python code. https://www.python.org/dev/peps/pep-0008/. [Online; accessed 21-December-2018].

Appendix A - IoF code structure

In this appendix, the structure of the IoF software is presented. For the full source code of the IoF system, see [20]. `.toml` files are the configuration files used by the back-end. The `metadata.toml` file must me formatted and placed manually, but the `metadata_conversion.toml` will be generated automatically, and so will the other `.toml`-, `.db`-, and `.log`-files. The structure of the back-end is following typical Python package structuring. The Dash application does not have a clear modular construction, but is following typical structure with an asset folder and resources. The source code can be summarized like this inside the root folder:

```
docs/
    build/
        html/
README.md
src/
    backend/
        .config/
            config.toml
            db_names.toml
            metadata_converison.toml
            metadata_positioning.toml
            metadata.toml
        dbmanager/
            databases/
                iof.db
                backup.db
```

```
            dbformat.py
            dbinit.py
            dbmanager.py
            msgbackup.py
            msgconversion.py
            pdop.py
            positioning.py
            tdoa.py
        initbackend.py
        logs/
            client.log
            main.log
        main.py
        mqttclient.py
        msghandler/
            conversion.py
            msghandler.py
            packet.py
            protocol.py
    frontend/
        app_plasma.py
        assets/
            base.css
            gallery-style.css
            general-app-page.css
            iof_icon.png
            github_mark.png
        db_to_plasma.py
        layoutCode.py
        live_app.py
```

```
        metaData.py
        plasma_state.pkl
venv/
```

Appendix B - TDOA Python code

In this appendix, the implemented TDOA python code is listed. None of the other positioning code is listed here.

```python
import numpy as np


def tdoa_hyperbola_algorithm(
    depth: float, tstamps: Timestamps, station_data: StationData
) -> CoordXYZ:
    """Calculates x-y-z coordinates based on station data, depth and timestamps.

    Calculates x-y-z coordinates with algorithm from Bertrand T. Fang's 1989 paper,
    | --> 'Simple Solutions for Hyperbolic and Related Position Fixes'.
    Algorithm is a 'Time Difference of Arrival' (TDOA) technique for positoning, where 3
    stations (A, B, C) with known positions, are used to measure difference in times of
    arrival of signal, leading to a hyperbolic solution for position fix. In this case,
    providing a 2D solution, while the third dimension z is provided by depth argument.

    Args:
        depth: Depth of signal, defining z-value of position [m units]
        timestamps: Dataclass 'Timestamps' with member variables containing
            second and millisecond time arrival of signal in station A, B and C.
        station_data: Dataclass 'StationData' with member variables containing
            integer id, latitude/longitude position, depth, and x-y-z position
            for station A, B, and C [m units].

            XY position of A is always (0, 0)
            XY position of B is always (b, 0)
            XY position of C is always (cx, cy)

    Returns:
        An instance of dataclass 'CoordXYZ' with member variables x, y, and z.
        Coordinates describe 3D-position of signal in relation to coordinate
        system defined by station data. The position is returned in unit cm.
    """
    # Extract values from input matrixes in algorithm/paper naming convention
    sec_a, msec_a = tstamps.sec_a, tstamps.msec_a
    sec_b, msec_b = tstamps.sec_b, tstamps.msec_b
    sec_c, msec_c = tstamps.sec_c, tstamps.msec_c

```

```
    # Coordinates of stations
    b = station_data.xyz_B.x
    cx = station_data.xyz_C.x
    cy = station_data.xyz_C.y
    c = np.sqrt(cx ** 2 + cy ** 2)
    z = depth - station_data.depth

    # Main equations:
    # | sqrt(x² + y² + z²) - sqrt((x-b)² + y² + z²) = V * T_ab = R_ab  (1)
    # | sqrt(x² + y² + z²) - sqrt((x-cx)² + (y-cy)² + z²) = V * T_ac = R_ac  (2)

    # Sound speed
    v = 1500

    # Calculate R_ab and R_ac using eq. (1) and (2)
    T_a = sec_a + (msec_a / 1000)
    T_b = sec_b + (msec_b / 1000)
    T_c = sec_c + (msec_c / 1000)
    R_ab = v * (T_a - T_b)
    R_ac = v * (T_a - T_c)

    # create x_candidates and y_candidates vectors
    x_candidates = np.zeros(2)
    y_candidates = np.zeros(2)

    # Transposing, squaring and simplifying equation (1) and (2) gives:
    # | R²_ab - b² + 2b*x = 2R_ab*sqrt(x²+y²+z²)  (3)
    # | R²_ac - c² + 2*cx*x + 2*cy*y = 2R_ac*sqrt(x²+y²+z²)  (4)

    if R_ab == 0:
        # equation (3) with R_ab equal to 0:
        # | ==> x = b/2
        x = b / 2
        x_candidates[0] = x  # equal roots
        x_candidates[1] = x  # equal roots
        if R_ac == 0:
            # If R_ab and R_ac is 0, position equal distance to all stations, trivial
            # Solve equation (4) for y with R_ab = 0 (x = b/2) and R_ac = 0:
            # | ==> y = (c² - 2*cx*x) / (2*cy)
            y = ((c ** 2) - 2 * cx * x) / (2 * cy)
            positions = np.array([CoordXYZ(x, y, depth)])
            return positions
        else:
            # Solve equation (4) for y with z = depth and R_ab = 0 (x = b/2):
            # | Put equation (4) on quadratic form:
            #   | term = R²_ac - c² + 2*cx*x
            #   | a_2 = 4*cy*(cy²-R²_ac)
            #   | a_1 = 2*cy*term
            #   | a_0 = term² - 4*R²_ac*(x²+z²)
            #   | ==> y²*a_2 + y*a_1 + a_0 = 0
            quad_term = (R_ac ** 2) - (c ** 2) + (2 * cx * x)
            a_2 = 4 * ((cy ** 2) - (R_ac ** 2))
```

```
            a_1 = 2 * (2 * cy) * quad_term
            a_0 = (quad_term ** 2) - (4 * (R_ac ** 2) * ((x ** 2) + (z ** 2)))
            poly = np.array([a_2, a_1, a_0])
            y_candidates = np.roots(poly)
    else:
        # Additional equations (equations for g, h, d, e and f given in code):
        # | y = g * x + h  (5)
        # | z = +-sqrt(d*x² + e*x + f)  (8)
        #   | ==> dx² + ex + f-z² = 0
        # | pos_xyz = xi^ + (g * x + h)j^ +- sqrt(d*x² + e*x + f)k^  (13)
        #   | where i^, j^, and k^ are unit vectors in x-, y- and z-direction.

        # Calculate g, h, d, e and f
        b_Rab = b / R_ab
        b_Rab_1 = 1 - (b_Rab ** 2)
        g = (R_ac * b_Rab - cx) / cy  # eq. (6)
        h = ((c ** 2) - (R_ac ** 2) + (R_ac * R_ab * b_Rab_1)) / (2 * cy)  # eq. (7)
        d = -(b_Rab_1 + (g ** 2))  # eq. (10)
        e = b * b_Rab_1 - 2 * g * h  # eq. (11)
        f = ((R_ab ** 2) / 4) * (b_Rab_1 ** 2) - (h ** 2)  # eq. (12)

        # Calculate x and y, where x is found with equation (8) and y with equation (5)
        x_polynomial = np.array([d, e, f - (z ** 2)])
        x_candidates = np.roots(x_polynomial)
        y_candidates = g * x_candidates + h

    # Reject complex solutions if they exist (does not reject complex number with 0j)
    index = np.argwhere(np.iscomplex(x_candidates))
    x_candidates = np.delete(x_candidates, index)
    y_candidates = np.delete(y_candidates, index)

    # Take the real value in case the candidate object has 0j imaginary part
    x_candidates = np.real(x_candidates)
    y_candidates = np.real(y_candidates)
    numOfCandidates = x_candidates.size

    # Pack candidates into array and return
    if numOfCandidates == 1:
        x = x_candidates.item()
        y = y_candidates.item()
        positions = np.array([CoordXYZ(x, y, depth)])
    elif numOfCandidates == 2:
        x_0, x_1 = x_candidates
        y_0, y_1 = y_candidates
        positions = np.array([CoordXYZ(x_0, y_0, depth), CoordXYZ(x_1, y_1, depth)])
    else:
        positions = None
    return positions
```

Appendix C - timestamp adjustment Python code

In this appendix, the implemented python code to adjust timestamps that have drifted in triplets is listed.

```
import pandas as pd

def _adjust_tstamp_drift_of_triplet(df: pd.Dataframe) -> List[pd.DataFrame]:
    """Return list of pandas DataFrames where timestamp offsets has been adjusted.

    Sorts dataframe based on timestamp, finds triplets where timestamp is equal +-2, and
    adjusts any timestamps +-2 from 2nd timestamp to be equal to 2nd timestamp. Returns
    a list of all valid triplets.

    Args:
        df: pd.DataFrame where columns "timestamp" and "millisecond" are used to adjust.

    Returns:
        Returns list of pd.DataFrame where timestamps offset +-2 from middle timestamp
        is adjusted. For example:

        | timestamp  | millisecond | frequency | tagID | tagData |
        | 1556555369 |     335     |     69    |   12  |   3.5   |
        | 1556555370 |     345     |     69    |   12  |   3.5   |
        | 1556555371 |     350     |     69    |   12  |   3.5   |

        becomes -->

        | timestamp  | millisecond | frequency | tagID | tagData |
        | 1556555370 |     335     |     69    |   12  |   3.5   |
        | 1556555370 |     345     |     69    |   12  |   3.5   |
        | 1556555370 |     350     |     69    |   12  |   3.5   |
    """
    ts_drift_threshold = 2
    ms_1km = 0.667

    # Sort dataframe by timestamps in case some timestamps are in the wrong order
    df = df.sort_values("timestamp")
    df = df.reset_index(drop=True)

```

```
    # Extract timestamps and find all triplets within dataframe
    ts = df["timestamp"]
    last_indices = ts.index[ts.diff(periods=2) <= ts_drift_threshold]
    all_indices = last_indices.append(
        [last_indices - 1, last_indices - 2]
    ).sort_values()

    # Mask out all detections that aren't triplets
    mask_values = [i for i in range(len(last_indices)) for _ in range(3)]
    df.loc[all_indices, "mask"] = mask_values
    df = df[df["mask"].notnull()]
    if df.empty:
        return []

    # Adjust timestamps that have drifted
    # | if 2nd timestamp in triplet is much larger than the 1st, add 2nd index to list
    # | if 3rd timestamp in triplet is much larger than the 2nd, add 2nd index to list
    df["drift"] = df.apply(lambda x: x["timestamp"] + x["millisecond"] / 1000, axis=1)
    drift = df["drift"].diff()
    drift_3rd = drift[last_indices].where(abs(drift[last_indices]) >= ms_1km)
    drift_1st = drift[last_indices - 1].where(abs(drift[last_indices - 1]) >= ms_1km)
    drift_indices = drift_3rd.dropna().index - 1  # -1 to get index of 2nd timestamp
    drift_indices = drift_indices.append(drift_1st.dropna().index)

    # Set timestamp 1 and 3 of each triplet with drift has equal to 2nd timestamp
    df.loc[drift_indices - 1, "timestamp"] = ts[drift_indices].values
    df.loc[drift_indices + 1, "timestamp"] = ts[drift_indices].values

    # get and return triplets as list of dataframes
    triplets = [v.drop(["mask", "drift"], axis=1) for _, v in df.groupby("mask")]
    # triplets = [v.drop(["mask"], axis=1) for _, v in df.groupby("mask")]
    return triplets
```