FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Mobile application for scanning identification of nearby drones in accordance with new EU regulations
Student:	Bc. Matej Glejtek
Supervisor:	Ing. Lukáš Brchl
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

New EU regulations require all future drones to broadcast their position and identification data to provide safety & security for the general public on the ground. The broadcasting technology is based on Wi-Fi and Bluetooth, thus all mobile phones can receive it. This thesis aims to implement a multiplatform application (Android and iOS) for the general public that will be able to scan those broadcasts and display them in a user-friendly way. Such an app is currently not available in the app stores.

- Research existing solutions for Direct Remote ID of drones and study its latest standards.
- Design and implement a standalone Flutter library that will scan Drone Remote ID broadcasts on both Android and iOS.
- Design and implement sample application using this library and try your best to release it into the App stores.
- Test the application with the general public and collect their feedback.
- Evaluate the resulting application and suggest its future improvements.

Electronically approved by Ing. Michal Valenta, Ph.D. on 8 February 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Mobile application for scanning identification of nearby drones in accordance with new EU regulations

Bc. Matej Glejtek

Department of Software Engineering
Supervisor: Ing. Lukáš Brchl

May 4, 2022

Acknowledgements

In the first place, I would like to thank my thesis supervisor Lukáš Brchl for his overseeing of the process and his support. Next, I want to appreciate the approach of the employees of the company Dronetag, especially Marián Hlaváč, who helped with the Flutter development. Last, but not least, I would like to thank all the volunteers that took part in the testing.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived its right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act. This fact shall not affect the provisions of Article 47b of the Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Prague on May 4, 2022 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Glejtek, Matej. *Mobile application for scanning identification of nearby drones in accordance with new EU regulations.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Cieľom tejto práce je navrhnúť a naimplementovať mobilnú aplikáciu na skenovanie bezpilotných lietadiel v okolí. Na základe nových regulácii sa piloti musia registrovať a drony musia vysielať svoju polohu a identifikáciu pomocou bezdrátových technologií. Aplikácia bude zbierať a zobrazovať tieto dáta pre užívateľov oboch hlavných mobilných platforiem. Riešenie pomôže sprehľadniť prevádzku dronov. Verejnosť získa možnosť identifikovať zodpovedné osoby v prípade, že bude ohrozená bezpečnosť alebo súkromie.

Klíčová slova drony, bezpilotné lietadlá, mobilné aplikácie, vzdialená identifikácia, bezdrátové technológie, multi-platform, open-source

Abstract

The thesis aims to design and implement a mobile application that will detect wireless broadcasts from unmanned aerial vehicles. According to new regulations, pilots need to register, and their drones are required to transmit the position and identification in a way that data can be acquired by mobile phones. The solution will gather data from the surroundings, giving users complete information about aerial traffic in their vicinity. The application will help to make drone operations safer because the general public may use it to identify operators and make them accountable for their actions in case the security or privacy is threatened.

Keywords drones, unmanned vehicles, mobile application, remote identification, wireless communication, multi-platform, open-source

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, the usage of unmanned aircraft is slowly becoming more common. Many people use drones just for leisure, but we can also see a significant rise in commercial usage. Even in the Czech Republic, companies are incorporating drones into their operation. For example, the company Kytary.cz is testing drone deliveries. According to Filip Černý, company manager, drones can cut their delivery cost to half [1]. The analysis predicts a further rise in the drone sector. Drone growth will occur across many segments of the enterprise industry: agriculture, construction and mining, insurance, media, and law enforcement [2]. This means that the amount of airspace traffic will increase, and there will also be a number of questions regarding safety and privacy.

The rising number of Unmanned Aircraft Systems (UAS) entering the airspace and increased complexity of operations of Unmanned Aircraft Systems (UAS) beyond visual line of sight, initially at a very low level, poses safety, security, privacy, and environmental risks [3].

Everyone understands why cars need license plates: drivers have to be accountable, and enforcement of the legislation must be possible. Like for cars, unmanned aircraft systems operations may create risks to other people's safety and result in damages and casualties. Therefore, drone operators must comply with safety regulations and be accountable for the damages they may cause. However, unlike cars, drones' ubiquity and the lack of transparency of their operation, as drone pilots and drones themselves may be difficult to identify, create several other risks. These risks relate to infringement of citizens' fundamental rights to privacy and data protection, to security risks associated with criminal and terrorist activities, or simply irresponsible behavior. They concern mainly operations conducted under 120m altitude with consumer drones that can be easily purchased on the Internet and are particularly prone to be used in all kinds of malicious or careless operations [4]. To ensure operator accountability, enable the enforcement of both safety and privacy legislation and contribute to addressing security risks, European Union (EU) and the United States of America (USA) impose the registration of drone operators.

The drone industry movement motivated policymakers to develop regulations that require pilots to be registered, and most importantly, drones will be required to broadcast information about themselves. The general public could then quickly identify the aircraft and its owner, as we can do with other vehicles, such as cars with their license plates. This thesis aims to create a universal tool for identifying aircraft in real-time. Such a tool has to collect data from many different aircraft types from any manufacturer, as long as the vehicle complies with the joint legislative framework. With the proposed solution, we will address the general public, for whom we will create a standalone mobile application. We also plan to make the solution open-source, so other developers can use it in their own projects and thus contribute to making the airspace safer and more transparent.

In the process, we will research the current solutions for Remote ID, investigate regulations and observe how they are being implemented. The solution will use wireless communication using radio waves, so understanding the technology and used protocols and standards is necessary. After the research part, we will focus on choosing the appropriate tools for implementing the solution and layout of the software system's architecture. The assignment determines that the Flutter library will be developed, as well as the mobile application on top of it. The software analysis stage consists of modeling the use cases, describing the application domain, and proposing the solution's architecture. After determining the software architecture and dependencies, we will move to implementation. During development, we will focus on creating a clean structure and applying software development practices and patterns best suited for the Flutter mobile platform.

We also intend to test the application with the target audience and collect relevant feedback. In the evaluation stage, we will assess the usability of the created application, as well as propose possible future development and extensions.

Chapter 2

Research

The research part of the thesis aims to familiarize ourselves with the concept of Remote ID, review wireless communication standards that use radio waves, learn about the protocols currently in place, and lastly, analyze existing solutions for Remote ID of unmanned aircraft. We will draw from this during the design and implementation stages.

2.1 Drone Identification Regulations

Current leaders in developing drone legislation and regulations are the USA and the EU. In both political entities, there are already approved regulations that administer the operations of Unmanned Autonomous Vehicle (UAV). Currently, we are in a period where the regulations are not enforced. In the EU, European Commission has adopted a *Commission Delegated Regulation 2019/945 of 12 March 2019 on unmanned aircraft systems and on third-country operators of unmanned aircraft systems* and *Commission Implementing Regulation 2019/947 of 24 May 2019 on the rules and procedures for the operation of unmanned aircraft.* In the USA, the Federal Aviation Administration (FAA), which is an agency within the Department of Transportation, published a *Federal Aviation Regulation, Rule 89.* Regulations provide a general framework for the operation of UAV without specifying technical details. We will mainly focus on European regulations.

The article that concerns are the most is article 6 of regulation *2019/945*: "Each Unmanned Aircraft (UA) intended to be operated in the "specific" category and at a height below 120 meters shall be equipped with a Remote ID system" [5]. Moreover, it requires a "periodic transmission of data in real-time during the whole duration of the flight, in a way that it can be received by existing mobile devices". To find out what is meant by the "specific" category, we have to take a look at regulation *2019/947*, and specifically, articles 4 and 5. The "specific" category is defined as any UAV operation that does not fulfill

the requirements for the "open" category. To fall into the "open" category, an operation must meet the following: the unmanned aircraft has a maximum take-off mass of less than 25 kg, the remote pilot ensures that the UA is kept at a safe distance from people and that it is not flown over assemblies of people, the remote pilot keeps the unmanned aircraft in Visual line-of-sight (VLOS) during flight, the unmanned aircraft is maintained within 120 meters from the closest point of the surface of the earth, the unmanned aircraft does not carry dangerous goods and does not drop any material [6]. If any part of the specification for the "open" category is not met, the UAS operator shall be required to obtain an operational authorization from the competent authority in the Member State where it is registered and thus belongs to the "specific" category and must also obey rules of Remote ID.

Regulations only state what is required, not how to accomplish the requirements. To meet the mentioned *European Commission Delegated Regulation 2019/945* and the *Commission Implementing Regulation 2019/947*, Aerospace and Defence Industries Association of Europe - Standardization (ASD-STAN) has developed the *prEN 4709-002* Direct Remote Identification (DRI) specification. ASD-STAN is an association that establishes, develops, and maintains standards on behalf of the European aerospace industry [7]. It is an Associated Body with European Committee for Standardization for Aerospace Standards. The final version of the standard has been published on their website [8]. It specifies Wi-Fi and Bluetooth broadcast methods for Remote ID that are compliant with the American *ASTM F3411 v1.1* specification. *ASTM F3411-19* is an American standard that accommodates rules of the FAA. Creators of the European standard state that "During the development of the ASD-STAN DRI standard, the emphasis was laid on creating a solution that would be compatible with the *ASTM F3411–19* [4]. Both of the mentioned standards have been defined to specify how UA or UAS can publish their ID, location, altitude, and other, either via direct Bluetooth or Wi-Fi broadcast or via an internet connection to a Remote ID server. There are certain differences in the European and American approaches, for example, which standard is optional and which is mandatory. Wi-Fi Beacon broadcasts are mandatory in Europe, but in the USA they are optional. Moreover, the EU regulation defines optional and recommended fields that bring some additional benefits in the EU environment.

There are ongoing discussions in International Organization for Standardization (ISO) organization to harmonize DRI globally [9]. We will mainly focus on the European regulation, but the data semantics are generally the same, so the created solution will be helpful on both continents.

In the USA, operators of UAS have thirty months since January 2021 to comply with the regulation, and manufacturers have 18 months after the publication date to comply [10]. In Europe, the timelines for the rules requiring manufacturers and drone operators to comply have fluctuated. Originally, the regulation *2019/945* states that it comes into effect just 20 days after publi-

cation. However, the most recent information that we were able to find says that UAS that would otherwise be in the ”specific” category are allowed to be used in the ”open” category for a transitional period ending on 1 January 2023 [11].

2.2 Regulation Implementation

The practical implementation of DRI on the UAS can be foreseen in two manners. For most of the drones already sold to customers or that are on the market right now, firmware update from UA manufacturers to fulfill the DRI requirements is expected. The UAS that will not receive this kind of support from manufacturers or those that do not have the appropriate Wi-Fi or Bluetooth hardware onboard still have a chance to retrofit by using add-on devices from 3rd party companies. An example of such a 3rd party company is the Dronetag [12]. Their device, called Dronetag Mini, ensures both Network Remote Identification (NRI) and DRI. It is designed as an add-on that can be mounted to any drone. Mini ensures that the drone is visible to all air traffic participants. Mini receives the drone's coordinates from Global Navigation Satellite System (GNSS), such as American Global Positioning System (GPS) or European Galileo. Then it sends the coordinates, along with the drone identification, in real-time to the central system through a mobile network or to everyone around via Bluetooth [13].

2.3 Remote ID and Semantic Model

The Remote ID function provides a means for an observer in the vicinity to retrieve this identification information without having physical access to the UA, using regular mobile devices with specific software downloaded from usual application online stores. The aim of Remote ID is to enable organized management of drone operations to ensure safe airspace. Today, if a pilot heads with his drone into a controlled perimeter around an airport, for example, it is not possible to contact him and warn him of the potential risk of collision with other aircraft - and this is one of the problems that Remote ID solves.

The rule essentially requires a “digital license plate” for UAV to be operated in the United States and the EU, one that both people on the ground and other airspace users can receive [10]. DRI means a system that ensures the local broadcast of information about a UA in operation. More specifically, we will address the drone's capability to be identified during the whole duration of the flight, in real-time and with no specific connectivity or ground infrastructure link, by existing mobile devices when within the broadcasting range. Such functionality, based on an open and documented transmission protocol, contributes to addressing security threats and supporting drone operators' obligations concerning citizens' fundamental rights to privacy and protection

of personal data. It can be used by law enforcement people, critical infrastructure managers, and the public to get instantaneous information on the drone flying around, providing various information such as UA serial number, UA navigation data, operational status, UAS Operator registration number and position as defined in the *Commission Delegated Regulation 2019/945*. Since the EU regulation *2019/945* requires DRI information to be broadcasted using an "open and documented protocol," this standard does not define technological measures to protect the confidentiality of the data broadcasted [4]. To operate drones in the open or specific category, a registration process is in place to register the owner of a UAS as Operator [4]. After registering as an Operator in one EU member state, the UAS operator will receive an Operator Registration Number, which needs to be uploaded as Operator ID information to the UA. The operator needs to register once, not for every vehicle he pilots.

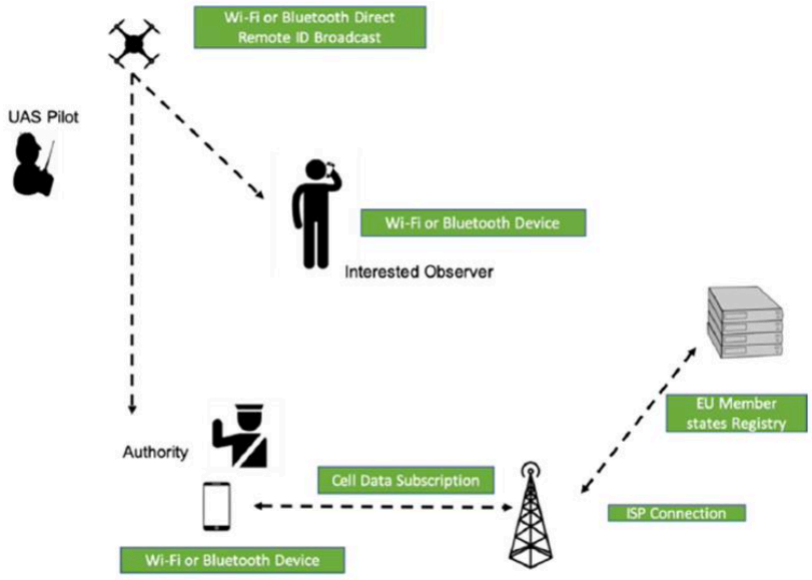

Figure 2.1: Remote ID Concept

2.3.1 Semantic Model

The structure of the data and the semantic model is based on a standard created by ASD-STAN, which precisely describes the message contents and various types of messages used for DRI. Some data fields are identified as mandatory to ensure compliance with the EU regulation, and other fields have been defined for enhanced operational benefits and interoperability with different standards. "The data fields defined in the data dictionary are assembled into messages following a defined structure, with a message header identifying the

message type and a block message containing the message payload. The content of each message payload, the syntax, and encoding rules of the data fields are specified" [4].

2.3.2 Message Types

The system broadcasts these types of messages:

- Basic ID Message - contains ID information for UA, ID type and type of UA
- Location Message - contains position, direction, speed, timestamp
- Self ID Message - allows operators to define their own messages, for example for mission purpose
- System ID Message - information about pilot location, aircraft grouping and other
- Operator ID Message - contains the Operator ID, which is the UAS Operator Registration Number
- Message Pack - several messages grouped into one

2.3.3 Operator Registration Number

The UAS operator ID is issued by the National authorities in each particular EU member state and is composed of two parts. The first part, known as a public part, is a sequence of 16 alphanumeric characters. The second part contains randomly generated secure characters and is known as a private part. The public part starts with three characters that define where the operator is registered (e.g., CZE for Czechia) and ends with one checksum character. Authorities who have access to the database of UAS Operator IDs can check if the public part is valid by accessing the private part and calculating the checksum.

The main purpose of the UAS Operator ID is to upload it to the DRI system. However, only the public part is stored in the drone and is broadcasted. The private part is not stored and is only used for DRI system validation. The operator should not tell their private part to anyone as they may misuse their identity. The example UAS Operator ID is "FIN87astrdge12k8" as public part and "xyz" as the private one. If the UAS Operator ID is not configured in the drone, it must broadcast the "NULL" string [9].

2.4 Remote ID Technology and Standards

In this section, we will describe technologies used for the Remote ID of drones through wireless communication. We will describe two leading technologies, Bluetooth and Wi-Fi, and the different versions both of these have. Before we even dive into the differences between Wi-Fi and Bluetooth, it's important to note that both of these technologies share a common foundation in the wireless electronics family through their use of radio waves. Radio waves are one of many types of waves in the electromagnetic spectrum, which includes other family members like x-rays, gamma rays, infrared rays, and more. These waves can all defy even the toughest physical barriers, transmitting data, video, audio, and more through the vacuum of space at the speed of light. Generally, the Wi-Fi solutions have a greater range due to the higher transmission power compared to the Bluetooth ones.

On this electromagnetic spectrum, one can measure and classify radio waves that are used in Wi-Fi, Bluetooth, and other applications in one of two ways: by frequency - this is the count of how many electromagnetic waves pass through a given point every second and is measured in Hertz, and by wavelength - this is the distance that we can measure between two of the highest points in a radio wave, which can range anywhere from 100 meters to 1 centimeter depending on the radio wave we are observing.

Within the radio wave family, there are distinct bands separated by both frequency and wavelength, providing specific channels for devices to use [14]. The standards we will study operate on one or more channels. From the figure 2.2, it is apparent that the receiving capabilities are lower for iOS phones, as just Bluetooth 4 support is claimed.

	Expected range	Maximum TX power	Android receive	Android update rate	iOS receive	iOS update rate
Bluetooth 4 Legacy Advertising	250 m	10 mW	All models	High	All models	High
Bluetooth 5 Long Range (Coded PHY S8)	1 km	10 mW	Selected newer models	High	None	None
Wi-Fi NAN	2 km	100 mW	Selected newer models	High	None	None
Wi-Fi Beacon	2 km	100 mW	All models	Low by default (see below)	Unknown	Unknown

Figure 2.2: Table of Technical Standards Used for Remote ID

2.4.1 Bluetooth

Bluetooth is a short-range wireless technology standard that is used for exchanging data between fixed and mobile devices over short distances using Ultra High Frequency (UHF) radio waves (300 MHz - 3 GHz) in the Industrial, Scientific and Medical (ISM) bands, from 2.402 to 2.48 GHz. Bluetooth was developed by the company Ericsson in the 1990s. It is mainly used as an alternative to wire connections, to exchange files between nearby portable devices, and connect cell phones and music players with wireless headphones. Bluetooth divides data into packets and transmits each packet on one of 79 designated Bluetooth channels. Each channel has a bandwidth of 1 MHz.

Bluetooth is managed by the Bluetooth Special Interest Group, which has more than 35,000 member companies in the areas of telecommunication, computing, networking, and consumer electronics [15]. This group manages releases of new standards in cooperation with its members.

Bluetooth 4

This standard is sometimes referred to as Bluetooth Legacy or Bluetooth Low Energy due to reduced power consumption. However, Bluetooth Low Energy is just a part of Bluetooth 4 specification. In fact, the Low Energy project started at Nokia under the name Wibree but would be incorporated into the next generation of Bluetooth [16].

In general, version 4 was slower. It topped out around 1 Mbps, but it was a lot more power-efficient, allowing for battery-operated accessories, such as fitness sensors and healthcare devices. They could work for years on a single coin cell battery.

Bluetooth 4 also extended the operating range to 100m and lowered the typical latency quite a bit [16]. Bluetooth 4 Legacy advertisements have up to 31 bytes, from which 25 bytes can be used for Remote ID data. Bluetooth Low Energy uses 2 MHz spacing, which accommodates 40 channels. Legacy advertising operates on channels 37, 38, and 39.

Bluetooth 5

Next came version 5.0 in 2016. It significantly improved the maximum range. That came at the cost of data speed, but at closer ranges, version 5.0 could double the rates of its predecessor, as it can transfer data with speed up to 2 Mbps [16]. Advertising extensions in Bluetooth 5 provide the capability to offload advertising data from the three traditional advertising channels to the full set of data channels for more frequency diversity [17]. The Extended Advertising technology of Bluetooth 5 allows transmitting up to 255-byte advertisements by transmitting the additional data on the non-advertising channels.

On the primary channel, we have a primary beacon packet. The pointer in the primary packets informs the receiver which secondary channel to read the second packet from. The pointer in the primary packet shall be broadcast on all three advertisement channels, with the secondary packet transmitted in the remaining channels.

According to the standard by ASD-STAN, when the extended advertising technology is used, the Remote ID data messages shall be grouped together and sent as a single *Message Pack*. A maximum of 9 messages shall be included in a single message pack.

2.4.2 Wi-Fi

Wi-Fi is a family of wireless network protocols, which are commonly used for local area networking of devices and Internet access, allowing nearby digital devices to exchange data by radio waves. These are the most widely used computer networks in the world. A common misconception is that the term Wi-Fi is short for "wireless fidelity". Nonetheless, Wi-Fi is a trademarked phrase that refers to Institute of Electrical and Electronics Engineers (IEEE) 802.11x standards. Wi-Fi originated in Hawaii in 1971, where a wireless UHF packet network called ALOHAnet was used to connect the islands. Later protocols developed in 1991 by AT&T called WaveLAN became the precursor to the IEEE 802.11 standards. The Wi-Fi Alliance was formed in 1999 and currently owns the Wi-Fi registered trademark. It specifically defines Wi-Fi as any wireless local area network products that are based on the Institute of IEEE 802.11 standards" [18].

Wi-Fi Beacon

The beacon frame is one of the management frames in IEEE 802.11 based networks. Beacons are relatively short, regular transmissions from Access Point (AP) with the purpose of informing user devices about available Wi-Fi services and nearby access points. It contains all the information about the network. Beacon frames are transmitted periodically. They serve to announce a wireless network's presence and synchronize the members of the service set. A Beacon frame comprises of IEEE 802.11 Media Access Control (MAC) header, body, and a frame check sequence. The body includes parameters such as time between two Beacon transmissions, timestamp, or Service Set Identifier (SSID).

Wi-Fi Neighbour Aware Network (NaN)

Neighbour Awareness Networking is a technical specification of the Wi-Fi Alliance. It is a standard well suited for the peer-to-peer exchange of data between groups of devices. Messages shall be encoded within the Service Discovery Frame based on the NaN specification. On the receiver, it does not require connecting to any specific wireless network since it utilizes the mechanism that simply listens for Wi-Fi broadcasts and makes the data available for display. It allows devices to find each other and communicate over Wi-Fi without an access point. Two hypothetical phones with NaN could find each other and connect without any additional software or configuration, allowing them to share data at high speeds. NaN doesn't require the use of GPS, cellular data, or the internet to link up. There's also a low-power connection mode that allows for sharing small bits of data like sensor readings [19].

2.5 Compatible Smartphones

One would assume that the smartphone manufacturers will provide us with the exact specification of their products, including which wireless technology is supported. Unfortunately, this is not the case with the support of the Bluetooth 5 and Wi-Fi NaN technologies. In most cases, we must find it out by an experiment and see whether the device can recognize data broadcasted with these technologies. The list of devices for which we already did the experiment with a list of supported technologies can be found on GitHub [20].

Even in the official document by ASD-STAN, it is stated that "Please also note that there are differences in the antenna/signal strength receive capabilities between different phone models. Although two different models are listed as capable of receiving a certain broadcast type, they are not necessarily able to receive the signals equally well at equal distances" [4].

2.6 Curently Existing Applications

There are several applications on the application stores that enable users to plan their flights and see restricted areas, but they are meant mainly for users with their own drone fleet or have limited functionality.

Another type of application found in the stores is one designed for a specific product by one company and is not usable for the general public, for example, Unifly. They have an application for their own device, not for every broadcasting agent, as we plan to have.

The final group of applications we were able to find provided information about UAV legislative and tell users whether they are allowed to fly in their

vicinity, but they are purely informative and do not provide any real-time data. The summary of existing applications follows.

- *Droneradar* - polish Android application, creators claim it complies with regulation *2019/945*. It is just in the Polish language, but from the description on Google Play [21], we concluded it only tells the users where they can fly. No scanning is implemented.
- *Dedrone Drone Scanner* - the company Dedrone offers solutions for airspace security. They provide hardware and software components for high-risk areas, such as airports or prisons. Their solution is not meant for the general public.
- *Unifly* - just for the iOS, offers Remote ID and drone tracking but works with just one device - the BLIP tracker [22].
- *Airmap* - contains zones and airspace rules, but mainly for the USA. It cannot scan for advertisements and can connect to supported DJI drones.
- *AirHub* - pre-flight planning and post-flight analysis, worldwide airspace rules and areas, detection of DJI drones.

To conclude, there is a lack of applications for the general public for direct identification of unmanned aircraft. As the law is currently not enforced, it is possible that more applications will pop up after the full implementation of the law. This gives us a unique window of opportunity to be the first one to deliver such an application to stores.

Chapter 3

Design

The purpose of this chapter is to propose and describe a solution that will be implemented during the realization. The proposed solution consists of two main parts. One is the Flutter library, and the other is an example mobile application that will showcase the usage of the library. The goal is to implement the solution for both leading mobile platforms - Android and iOS. Flutter library will serve as a backend for receiving and parsing incoming data. The library has a common part written in a Dart language, which communicates with the native code part, which is different for iOS and Android and written in Swift and Kotlin, respectively. The native components will implement a common interface. Data collected in the form of Bluetooth or Wi-Fi advertisements will flow through the layers in the form of asynchronous messages. The example application will gather data from the library and present them to a user on a map. This chapter will describe each part of the solution in detail and explain what technological tools were used. We will take a look at relevant features of Dart language and explain the concept of reactive programming. It is also crucial to understand how the development of mobile applications in Flutter works, so we will study the framework features in-depth. Finally, we will design the example application, model use cases, and scenarios and propose the future look of the GUI.

3.1 Architecture

From the architecture point of view, the overall solution consists of three main layers - the client application layer, the Flutter library layer, and a layer of native platform code. Layers communicate just with the one directly above or under. The upper layer uses the interfaces offered by lower layers in the form of asynchronous method invocation. On the other hand, the data flows from the bottom layer to the presentation layer in the form of asynchronous messages, to which the upper layer subscribes. Data are structured into high-level messages,

defined just once for the whole system, and injected into every layer by the Pigeon library. The library ensures the correct format and that data types are compatible, as different programming languages may be used on every layer. Pigeon library also contains a mechanism for declaration of platform Application Programming Interface (API), that will the Flutter library invoke on the native platform. This approach simplifies the implementation as we do not need to have model classes in every language. The main parts of the system are shown in the following figure 3.1, as well as data flow. The next section will look at technologies used on every layer in detail.

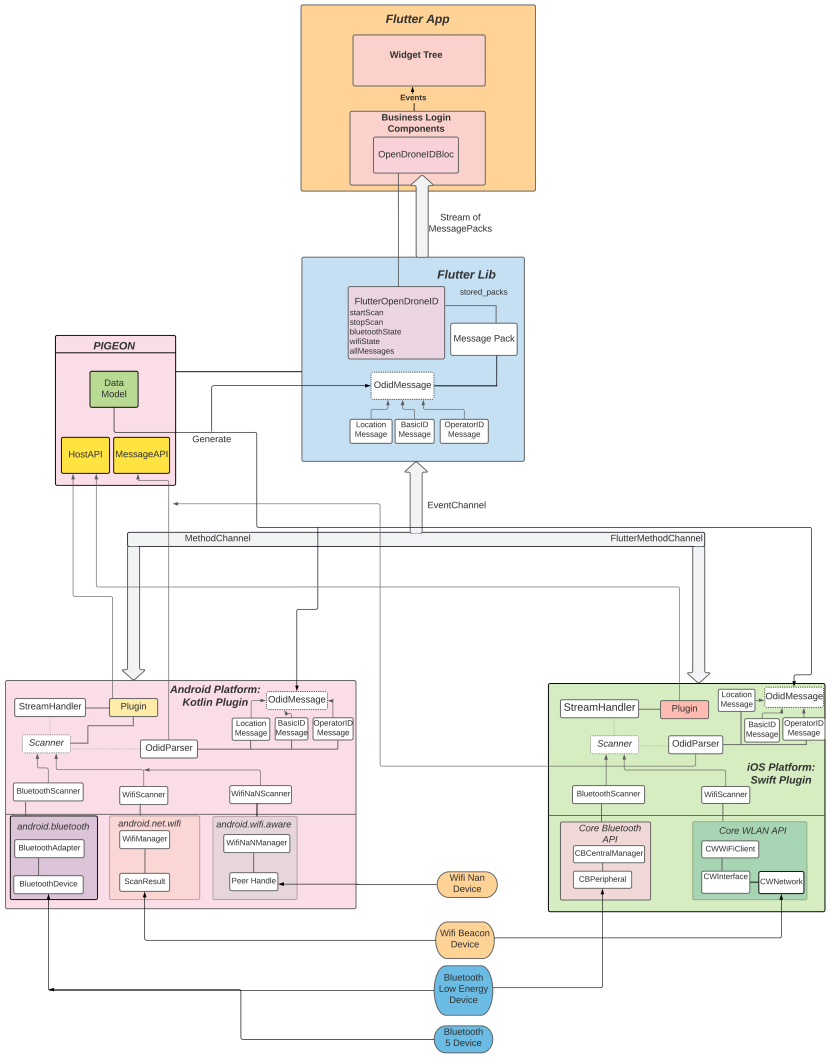

Figure 3.1: System Architecture Layout

3.2 Used Technology and Development Tools

The leading technology tools used to implement the assignment are the Dart programming language and the Flutter Software Development Kit (SDK). It was chosen because of the multi-platform approach and relative ease of use. In the following paragraphs, we will introduce the framework for creating the mobile application as well as native iOS and Android software components that will manage the Bluetooth and Wi-Fi connectivity. We will also highlight essential concepts used in Flutter mobile development, such as state management.

3.2.1 Dart Language

Dart is a programming language designed for client development, such as for the web and mobile apps, but it also can be used to build server and desktop applications [23]. It was developed by Google and first published in 2013. Dart is an object-oriented, class-based, garbage-collected language with C-style syntax. Other interesting features include null-safety, which aims to protect the programmer from null exceptions at runtime through static code analysis [24]. Dart can compile to either native code or JavaScript. This means that with Dart, we have just one codebase compiled to the target platform, for example, Apple's Swift Project. "Dart also forms the foundation of Flutter. Dart provides the language and runtimes that power Flutter apps, but Dart also supports many core developer tasks like formatting, analyzing, and testing code" [23].

Asynchronous and Reactive Programming

Asynchronous Programming is a principle that allows a portion of programs to run while waiting for some other activities to occur in the application thread. It will enable portions of code to run independently from the main workflow [25]. Reactive programming is programming with asynchronous data *streams*. A data *stream* is an object that emits multiple pieces of data over time. By subscribing to a *stream*, we can listen to all of the changes that happen. As long as we are subscribed to it, we'll get notified every time there's a new piece of data that's been added to the *streams*. We will utilize this approach when developing the solution. The source of data will be the native plugins receiving advertisements, and data will flow asynchronously towards the client application.

In Dart, there are a series of ways to write asynchronous code. To mark a method as asynchronous, we can use the *async* keyword. There are two classes for handling waiting for the result of the asynchronous operation - *Future* and *Stream*. A *Future* can provide only a single result over time — either an error or data that it delivers asynchronously. *Streams*, on the other

hand, can provide zero or more values or error results over time. They can push multiple pieces of data at different periods of time [26].

3.2.2 Flutter Library and Architecture

Flutter is an open-source User Interface (UI) software development kit created by Google. It is used to develop cross-platform applications for Android, iOS, Linux, macOS, and others, and the web, all from a single codebase [27] using a single language. Applications could be distributed through native application shops. First described in 2015, Flutter was released in May 2017. To cite the creators, "Flutter aims to provide a framework and tooling for creating user experiences without compromising any device or form factor. The Dart-powered Flutter engine supports fast development with stateful hot reload, and fast performance in production with native compilation, whether it is running on mobile, desktop, web, or embedded devices" [28].

Flutter follows several principles, one of which is the UI as code principle. This means that no visual editor is needed. Developers build a widget tree in a programming language. Implementation of standard functionalities is available in the form of packages, which can be found on *pub.dev* portal [29]. Packages are helpful for repeating tasks for which other developers created already verified solutions.

The SDK is made out of several components.

- Dart platform - the Dart language itself.
- Flutter engine - written primarily in C++, provides low-level rendering support and it interfaces with platform-specific SDKs such as those provided by Android and iOS [30].
- Foundation library - provides basic classes and functions, such as APIs to communicate with the engine [30].
- Design-specific widgets.
- Flutter Development Tools.

An integral part of the Flutter project is the *pubspec.yaml* file, used for dependency management and importing packages, fonts, images, and other assets. Code gets compiled into native apps. A developer has complete control of every pixel on the screen. Internally, Flutter does not use platform primitives - it does not compile into iOS/Android UI components.

Widgets

For building the UI with the Flutter SDK, we can use a vast collection of widgets. Widgets are building blocks of the application. They are basically pieces of code that describe what their view should look like, given their current configuration and state. A complex widget is composed of already existing smaller widgets. When a widget's state changes, the widget rebuilds its description, which the framework differentiates against the previous description in order to determine the minimal changes needed in the underlying render tree to transition from one state to the next [31]. There are many various types of widgets like layouts, containers, input and output widgets, and many more. All widgets can be found in the catalog [32].

We can also create our own widgets by subclassing *StatessWidget* or *StatefulWidget*, depending on whether the widget manages any state. A widget's main job is to implement a *build()* function, which describes the widget in terms of other, lower-level widgets [31].

Stateless widgets receive arguments from their parent widget, which they store in final member variables. When a widget is asked to build itself, it uses these stored values to derive new arguments for the widgets it creates.

But this is not sufficient, as applications typically carry some state. Flutter uses *StatefulWidgets* to capture this idea. *StatefulWidgets* are special widgets that know how to generate *State* objects, which are then used to hold state. Widgets are temporary objects used to construct a presentation of the application in its current state. State objects, on the other hand, are persistent between calls to *build()*, allowing them to remember information [31].

There are also separate widgets tailored to mobile platforms. For Android, we have the *Material* widgets library, and for iOS, the *Cupertino* widget library.

3.2.3 State Management

State, as the key concept in the Flutter SDK, means the data that affect the UI. Application is rebuilt according to the change of state of widgets. The *build()* method of a widget is called by the Flutter engine when its own state changes. This also means that all the child widgets of this widget will have to be instantiated again. Child widgets need data. Managing data in a top widget leads to a chain of arguments and, importantly, unnecessary rebuilds. For the application-wide states that affect several widgets, we will use the *Bloc* package [33] and subsequently the Bussiness Logic Components (BLoC) pattern [34], which separates the state from the widgets.

Bloc Package Approach

BLoC is a state management system for Flutter recommended by Google developers [35]. The idea is to have a global central store, one or several classes containing the data and thus the application's state, separated from the UI. The storage will never have any reference to the widgets on the UI screen. The UI screen will only observe changes coming from the BLoC class [35]. A state management library called *Bloc* was created and maintained by Felix Angelo [33]. It helps developers implement the BLoC design pattern in the Flutter application [36].

We need to use the *MultiBlocProvider* widget as the topmost application widget and register our created data storage classes. The root then provides one instance of data class, and the widget that wants the data can listen to the changes in data. *MultiBlocProvider* is a Flutter widget that creates and provides a BLoC to all of its children. This is known as a dependency injection widget so that a single instance of BLoC can be supplied to multiple widgets within a subtree. In other words, the entire subtree will benefit from a single event of a BLoC injected into it. Hence the subtree will be dependent on the BLoC we're providing [36].

3.2.4 Responsivness and Adaptivness

Responsiveness is the ability to handle different screen sizes and portrait and landscape mode changes. On the other hand, GUI is adaptive when it can adjust itself according to different Operating System (OS). For adaptivity, we can use widgets designed specifically to match the styles of target platforms: *Cupertino* for iOS or *Material* for Android.

For responsiveness, we will need to know which orientation and screen size are currently used. For example, we will lay out our aircraft detail into two columns for landscape and one column for the portrait. For this purpose, we will use Flutter *MediaQuery* class. From the *MediaQuery* we can get information about the current device size, as well as user preferences so that we can design your layout accordingly.

MediaQuery provides a higher-level view of the current app's screen size and can also give more detailed information about the device and its layout preferences. It can simply be accessed by calling *MediaQuery.of* in the *build* method of every widget [37]. This class will also help with adjusting the UI to different screen sizes, as we can get the exact width and height of the screen or even the size of the system status bar. We will define the sizes of components mainly using fractions of the total dimensions available.

3.2.5 Permissions

Not every Android phone supports newer technologies like Bluetooth 5 or Wi-Fi NaN. Therefore the application needs to verify which standard is supported on a device. This task will be carried out by the user and logged into our server to keep a database of smartphones and their supported standards for future use.

Furthermore, if the standard is available, the application needs permission from the user to access it. For this task, another library called *Permissions Handler* will be used. In most operating systems, permissions aren't just granted to apps at install time. Rather, developers have to ask the user for them while the app runs. This plugin provides a cross-platform API to request permissions and check their status [38]. During the first start of the application, the application will ask the user to enable protected features using this library. Namely, the application inevitably needs to use Wi-Fi and Bluetooth. Without them, it would be useless. Also very important, but not crucial, is the location permission, as the application can be used even without knowing the user's location. Lastly, if a user wants to export data to Comma Separated Values (CSV), we also need to ask for permission to access storage.

3.2.6 Writing Native Code

Flutter alone does not support many things like geolocation, payments, and video calling. If we want to use these kinds of advanced features in the Flutter project, we have to write our own plugin in the native code of Android and iOS. We need to use platform-specific functionality available through existing native packages to implement these tasks. This means that we will have a separate codebase for every platform. The platform-specific code will make up the Flutter Plugin. Flutter plugin is the wrapper of the native code [39].

Flutter allows us to call platform-specific APIs available in Java or Kotlin code on Android and in Objective-C or Swift code on iOS [40]. We will need to write the scanning using Wi-Fi or Bluetooth with native code. From Flutter, we have to send messages to a host on iOS or Android parts of the application over a platform channel. The host listens on the platform channel and receives the message. It then uses any platform-specific APIs using the native programming language and sends back a response to the Flutter portion of the application.

Platform Communication and the Pigeon Library

One of the most critical parts of the proposed software system is the communication between the Flutter application and native Android or iOS code. For this purpose, there exists a concept of method and platform channels.

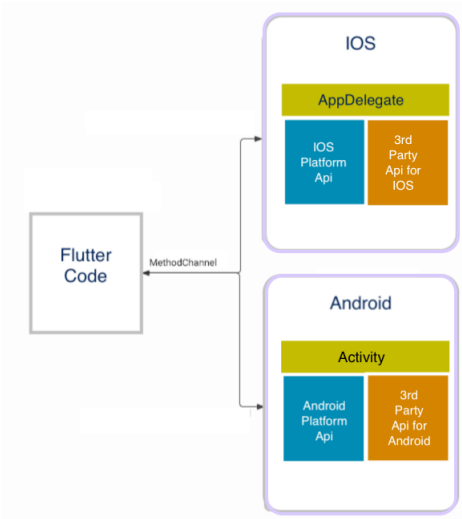

Figure 3.2: Schema of Native Code Invocation

Messages and responses are passed using these channels asynchronously to ensure the UI remains responsive.

On the client application side, *MethodChannel* enables sending messages that correspond to method calls. On the platform side, *MethodChannel* on Android and *FlutterMethodChannel* on iOS enable receiving method calls and sending back a result. Calling and receiving messages depends on the host and client declaring the same arguments and data types in order for messages to work. The *Pigeon* package can be used as an alternative to *MethodChannel* to generate code that sends messages in a structured and typesafe manner [41].

Using *Pigeon* eliminates the need to match strings between host and client for the names and datatypes of messages. The generated code is readable and guarantees there will be no conflicts between multiple clients of different versions. Supported languages are Objective-C, Java, Kotlin, and Swift [41].

On the other hand, the *Pigeon* library also has certain limitations. For example, declared methods can only use primitive datatypes or data types declared in a schema. Another significant limitation is that Pigeon does not support polymorphism. This will negatively affect architecture as we will need to handle each message type individually and cannot use common channels for them. This fact was observed during the initial phases of development. We could not find much information about the issue, but we found bug reports on forums confirming the problem. It may be solved by developers soon.

3.2.7 Compilation

Flutter apps don't directly compile to native Android and iOS apps. Instead, they run on the Flutter rendering engine (written in C++) and Flutter

Framework (written in Dart, just like Flutter applications), both of which get bundled up with every application. Then the SDK generates a package that's ready to go on each platform. We get the application, a new engine to run the Flutter code on, and enough native code to get the Flutter platform running on Android and iOS [27]. Cross-platform development does not mean Flutter applications will feel out of place on Android or iOS devices, as the resulting UI can be tailored for each platform.

3.2.8 Android Platform

The Android SDK contains Android Platform APIs and libraries that are used to implement various functionality, including wireless communication. The available API is documented on the Google Developers webpage [42]. If this would not be sufficient, we can also use 3rd party libraries.

Bluetooth 4 & 5

On the Android platform, we will utilize the library called *android.bluetooth*. Using the Bluetooth API, an application can perform the scan for other Bluetooth devices with the ability to read the advertisement data [43]. We do not need to establish a paired connection. We just need to read the advertisements. To achieve this, we can use the *BluetoothAdapter* class, which handles scanning and outputs result in the form of an instance of Bluetooth device, which holds all the data we need.

Wi-Fi

There are separate packages for the Wi-Fi Beacon and Wi-Fi NaN on the Android platform. The primary way to access Wi-Fi connectivity is to use the services of *WifiManager* API. We can use it to get a list of Wi-Fi access points that are visible from the device and also read the Beacon advertisements [44]. For the Wi-Fi Aware standard, which is just another name for Wi-Fi NaN, Android offers a library called *Wi-FiAware* [45]. *Wi-FiAware* capabilities are available just for devices running Android 8.0 or higher. According to Google Developers, Wi-Fi Aware network connections support higher throughput rates across longer distances than Bluetooth connections. The API has a mechanism for finding other nearby devices, which we can use.

3.2.9 iOS Platform

Generally speaking, the iOS platform lacks certain functionality and the ability to use Android's third-party packages. Citing the guidelines for releasing software to the AppStore, "Apps may only use public APIs" [46]. This means we can only use APIs that are known to Apple and documented on their

documentation site. Of course, this fact currently limits the technology we can use. Details will follow in the following sections. This situation may change in the future. In the meantime, we have to make do just with the Bluetooth 4.

Bluetooth 4

For implementing reading Bluetooth 4 advertisements, we will use the *CoreBluetooth* framework, which Apple provides. According to official documentation, [47], the *CoreBluetooth* framework supplies the classes needed for iOS and macOS applications to communicate with devices that are equipped with Bluetooth low energy wireless technology. Bluetooth low energy is based on the Bluetooth 4.0 specification, which, among other things, defines a set of protocols for communicating between low energy devices. Therefore, this package cannot be used for Bluetooth 5.

Wi-Fi

According to our research, there is currently no official framework supported by Apple that would enable scanning Wi-Fi networks. There is the *CoreWLAN* framework available. In the documentation [48], authors state that the
CoreWLAN framework provides API for querying wireless interfaces and choosing networks. Unfortunately, this framework is not available for the iOS operating system, just for macOS. Apple simply does not provide any simple way to scan the networks on mobile phones, presumably for security reasons. There are alternative frameworks that offer the needed functionality, but using them will cause Apple to disallow our application from the App Store.

3.3 Example Application

The application will be built using the Flutter SDK and its widgets library. Like the Flutter SDK determines, the application will have a tree-like widget structure, with the application itself as the central widget and all the other widgets incorporated into it. Low-fidelity wireframes were created in the initial stages to test the proposed UI concepts. The application UI will consist of the main page with a map showing user location, aircraft, and zones in proximity and one settings page with customizations of the application's behavior and additional information. Understanding users' needs and expectations help with creating more usable applications. In this section, we will model user interaction with the application. First, by defining the goals that can be

achieved by using the software, then we will model the use cases. Lastly, we will write down all the tasks that can be performed in the application.

3.3.1 User Goals

- See UAVs around the user.
- See details of a specific UAV.
- See restricted zones around the user.
- Display a legislation regulating drone operations.
- Show details of selected zone.
- Export gathered data in a CSV format.

3.3.2 Use Cases

The application will have the following use cases from the user's point of view:

- User expects to see a map with his own location and the location of aircraft near him. The user also supposes that the map will be customizable, and the user can center either on specific aircraft or on itself. Relevant data about detected aircraft will be presented to the user.
- In a detail of specific aircraft, the user would want to see the exact location, height, direction, and pilot identification number. In a full-detail view, the user expects all the possible information that can be gathered, like exact timestamp, location history, operator position, or operation description.
- Users expect to see zones marked on a map, with map location the shape outlined on the correct location. The zone should be clearly visible, and different types of zones may be highlighted with color.
- There should be an easy way to export or shared gathered data.
- Users expect an easy way to browse current legislation and rules regulating drone operation.

3.3.3 Task List

During the work with our application, users will be able to perform the following tasks.

- Show application tutorial.
- Start and stop scans.

- Choose which technology will carry out the scan. It could be Bluetooth, Wi-Fi, or both.
- Display a map of aircraft near me.
- Change map settings.
- Display a user's location on a map.
- Center map on the location of the user.
- Center map on the aircraft.
- Follow a trajectory of a flight of the selected aircraft.
- Display restricted zones on a map.
- Show list of nearby aircraft.
- Sort list of nearby aircraft according to distance from user or time.
- Show aircraft details.
- Show zone details.
- Export messages from one or all aircraft to CSV.
- Hide details and return to map.
- See which standard is supported on a certain device.

3.3.4 Wireframes

Our graphical interface is inspired by the known and well-established layout of almost all the applications providing maps and navigation, for example, *Google Maps Application* [49]. The reason for this choice is that we will match users' mental models, and the usability should be higher because users will not have to learn to use the application extensively.

The main page contains the map with a search bar at the top and a toolbar with map settings on the side. An active marker will pop up on the map when an aircraft is detected. On click on this marker, aircraft detail will slide up from the bottom of the screen, which can be maximized to see all the available data. When no aircraft is selected, the panel will contain a list of detected devices and zones. Icons with map settings, such as centering and moving through the map, are placed in a traditional position on the bottom right side of the screen, coupled with a button to start or stop scans.

We have created low-fidelity wireframes for a quick outline of the application structure. A wireframe is a schematic, a blueprint, useful to help programmers and designers think and communicate about the structure of

the software [50]. Since it is only a matter of determining the position of components, there is no need to take aesthetic details into account [51]. With the help of a wireframe model, we can establish the functional and logical structure of the GUI without concentrating on details. In the later stages, this basic structure won't change. In the following figure 3.3, we can see the map page with minimized slider, maximized slider with detail, or a list with device cards.

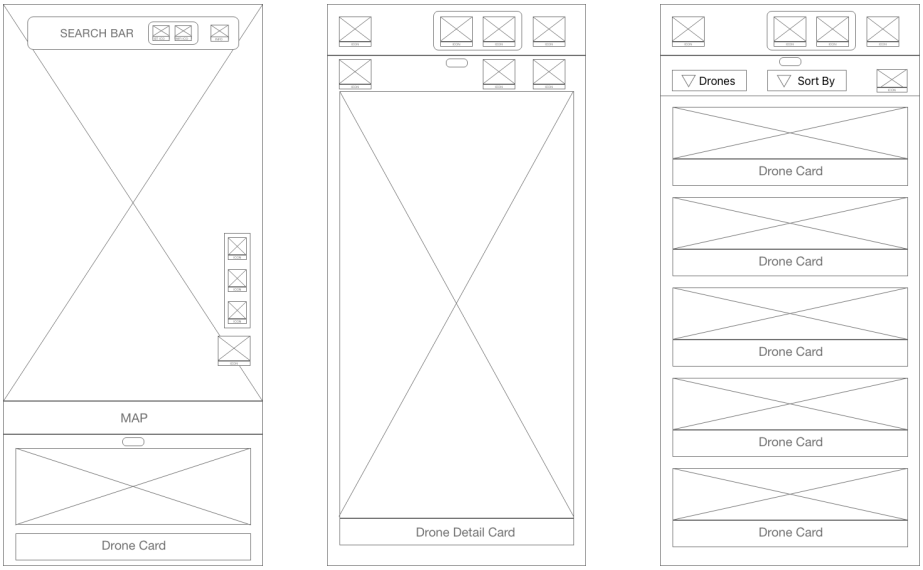

Figure 3.3: Wireframes

Chapter 4

Implementation

Following chapter reports on the process of implementing the requirements specified in the previous chapter. We will describe a whole implementation procedure, highlighting the interesting and essential parts and explaining used concepts. The realization stage of the application consists of creating the Flutter library, serving as the interface for underlying native code, where most of the heavy lifting occurs. Native code will use APIs introduced in a design chapter. The library usage is then showcased in the example application, developed with the help of Flutter SDK. We will sum up which widgets were used and how the widget tree functions. Implementation of important functionality will be elaborated in detail and shown in code snippets.

4.1 Flutter Library Implementation

The library is the most critical part of the solution, as it will carry out the key task of gathering the data from nearby aircraft. As described in the design chapter, the library is structured into three parts - Flutter code and native iOS and Android code, with a common data model. In the following subsection, we will describe the implementation of all the library parts.

4.1.1 Data Model

The data model of the application consists of different types of messages. Contents and semantics are given by standards explained in the research chapter. As stated in the design chapter, the *Pigeon* library is used to simplify the communication between the Flutter and native code. As the first step in implementation, we need to specify what data will be transferred between the agents. For this purpose, we created a file *schema.dart*. In this file, we declare all the possible messages and their fields.

For example, we have the *BasicID Message*.

```
class BasicIdMessage {
  late final int receivedTimestamp;
  late final String macAddress;
  late final MessageSource source;
  // signal strength
  late final int rssi;
  /// The primary identifier of UAS
  late final String uasId;
  /// Identification type
  late final IdType idType;
  /// Type of the aircraft
  late UaType uaType;
}
```

Listing 4.1: *BasicID Message* declaration.

4.1.2 Platform Interface

The *Pigeon* library also provides a mechanism for invoking host platform code. In *schema.dart*, we also declare API, which will be implemented separately for each platform and later will be invoked from Flutter. The API is shown on the following code snippet. The schema is declared in a Dart language. The *Pigeon* engine takes the schema as input. We will run the *Pigeon generate* utility on this file to generate code in the programming language of each platform. It generates Dart, Objective-C, and Kotlin code, which will be used in the Flutter library, iOS platform module, and Android platform module, respectively.

```
@HostApi()
abstract class Api {
  @async
  void startScanBluetooth();
  @async
  void startScanWifi();
  @async
  void stopScanBluetooth();
  @async
  void stopScanWifi();
  @async
  void setAutorestartBluetooth(bool enable);
  @async
  bool isScanningBluetooth();
  @async
  bool isScanningWifi();
  @async
  int bluetoothState();
}
```

Listing 4.2: Host Platform interface declaration.

Furthermore, we will declare a second interface with methods for parsing the incoming data into messages. The interface will also be implemented by native code but will be called just internally inside the platform module.

```
@HostApi()
abstract class MessageApi {
  int determineMessageType(Uint8List payload, int offset);
  BasicIdMessage fromBufferBasic(
      Uint8List payload, int offset, String macAddress);
  LocationMessage fromBufferLocation(
      Uint8List payload, int offset, String macAddress);
  OperatorIdMessage fromBufferOperatorId(
      Uint8List payload, int offset, String macAddress);
  // ...
}
```

Listing 4.3: Platform interface for data parsing.

4.1.3 Flutter Library Implementation

An essential task of the library is to start the scans, gather data and then provide them to subscribed listeners. Incoming messages are structured into message packs. The message pack consists of different messages from the same MAC address, thus one device. For invocation of platform methods, we will call methods of our Pigeon interface. The data will flow from the platform to the Flutter library in the form of *EventChannels*.

The common part of a Flutter plugin consists of main class, *FlutterOpenDroneId* and one model class, *MessagePack*. The *MessagePack* is a collection of messages from one source, it contains one message of one type, but not all message types need to be present.

Next, we will look at the *FlutterOpenDroneId* class. Here, we instantiate our Pigeon API. The Pigeon library will ensure that the correct implementation will be injected, and we can invoke methods to manage to scan. Then, we create *EventChannels* for every message type. It would be nicer if we had just one stream and messages had a common ancestor class. Still, as we explained in previous chapters, the Pigeon does not allow polymorphism, so we need separate channels for every message type.

In the *startScan* method, we initialize our subscriptions to event channels, and register listeners to data passed to the channel. When a new message arrives, we add it to the message pack associated with the device's MAC address and store the updated package. So, to sum up, we have one message pack for one device, and each message pack contains the last message of every type. The *EventChannels* carry binary data, but we do not need to worry about it because the Pigeon generated *decode* methods for every message type that we can use to convert the data into message classes.

We need to pass the received data further. For this task, we create a stream into which we will insert received messages grouped into message packs. "A stream is a sequence of asynchronous events. It is like an asynchronous Iterable—where, instead of getting the next event when the programmer asks for it, the stream tells the programmer that there is an event when it is ready" [52]. Every time a new message arrives, we update the device's message pack and send it to the stream. The library does not store the whole history of gained messages, it just passes it to a stream. Consumers get access to the stream by calling *allMessages* getter and setting up their own listeners.

```
class FlutterOpenDroneId {
  static late pigeon.Api _api = pigeon.Api();
  static const _locationMessagesEventChannel =
      const EventChannel('flutter_location_messages');
  // ... event streams for other messages follow
  static StreamSubscription? _locationMessagesSubscription;
  // ... subscriptions for other messages follow

  static Map<String, MessagePack> _storedPacks = {};
  static final _packController =
        StreamController<MessagePack>.broadcast();

  static Stream<MessagePack> get allMessages
     => _packController.stream;

  /// Starts scanning for nearby traffic
  static Future<void> startScan(usedTechnologies) async {
    _locationMessagesSubscription =
        _locationMessagesEventChannel
        .receiveBroadcastStream().listen((data) {
      final message = pigeon.LocationMessage.decode(data);
      if (message == null) return;
      _updatePacksWithLocation(message);
    });
    // ....
    if (usedTechnologies == UsedTechnologies.Bluetooth ||) {
      await _api.startScanBluetooth();
    }
    if (usedTechnologies == UsedTechnologies.Wifi) {
      await _api.startScanWifi();
    }
  }

  static Future<void> stopScan() async {
    await _api.stopScanBluetooth();
    await _api.stopScanWifi();
    _locationMessagesSubscription?.cancel();
  }
```

Listing 4.4: Main class of our Flutter library (1st part).

```
  static void _updatePacksWithLocation(
        pigeon.LocationMessage message) {
    final mac = message.macAddress as String;
    final storedPack =
        _storedPacks[message.macAddress] ??
          MessagePack(macAddress: mac);
    _storedPacks[mac] = storedPack.updateWithLocation(message);
    _packController.add(_storedPacks[message.macAddress]!);
  }
}
```

Listing 4.5: Main class of our Flutter library (2nd part).

4.1.4 Native Plugins Implementation

The structure of the native code is very similar on both platforms. We have the main plugin file, which implements our *HostAPI*. Then, for each standard, we have scanner classes. Instances of these classes then parse the data using parsers, like the *OdidMessageHandler.kt* on Android, implementing our *MessageAPI*. The data model is injected into both plugins from Pigeon, so we include the generated files in our code and can use them. We will take a look at implementation details in the following sections.

4.1.5 Android Native Code

The Android plugin is written in a Kotlin language. The main class is the *FlutterOpendroneidPlugin* class. It implements *FlutterPlugin* , *ActivityAware*, and our own *Pigeon.Api*. FlutterPlugin is the Interface to be implemented by all Flutter plugins. Our plugin also needs to react to Activity lifecycle events, e.g., *onCreate()*. Any such plugin should implement *ActivityAware* interface [53]. Important tasks are executed during attaching to the engine: we need to establish a connection to event channels, to which we will write data. For every channel, we create a *StreamHandler*, which we then pass to scanner classes. Scanner classes will use the handlers to pass the messages to our main library class. As events are passed in the form of binary data, we serialize our messages using the *toMap* method, generated by *Pigeon*. Lastly, for Wi-Fi scans, we decided to use *android.net.wifi.WifiManager* and *android.net.wifi.aware.WifiAwareManager*. As stated in the documentation, these objects should only be obtained from an application context [54], not instantiated. Therefore, we request the managers from context and pass them to scanners.

The following code examples contain simplified extracts of implementation to illustrate used concepts. Repeating and non-essential parts were omitted to preserve readability.

```
class FlutterOpendroneidPlugin
        : FlutterPlugin, ActivityAware, Pigeon.Api {
    private val locationStreamHandler = StreamHandler()
    // declarations of stream handler for every message type follow
    private var scanner: BluetoothScanner =
            BluetoothScanner( locationStreamHandler, /*...*/)
    private lateinit var wifiScanner: WifiScanner
    private lateinit var wifiNaNScanner: WifiNaNScanner

    override fun onAttachedToEngine(
        flutterPluginBinding: FlutterPlugin.FlutterPluginBinding
    ) {
        Pigeon.Api.setup(flutterPluginBinding.binaryMessenger, this)
        StreamHandler.bindMultipleHandlers(
          flutterPluginBinding.binaryMessenger,
          mapOf(
                "flutter_location_messages" to locationStreamHandler
                  //...
                )
        )
        context = flutterPluginBinding.applicationContext
        val wifiManager: WifiManager? =
                context.getSystemService(Context.WIFI_SERVICE)
        val wifiAwareManager: WifiAwareManager? =
                context.getSystemService(Context.WIFI_AWARE_SERVICE)
        wifiScanner =
                WifiScanner(
                        locationStreamHandler,
                                /*...*/, wifiManager)
        wifiNaNScanner =
                WifiNaNScanner(
                        locationStreamHandler,
                        /*...*/,  wifiAwareManager)
    }

    @RequiresApi(Build.VERSION_CODES.O)
    override fun startScanBluetooth(result: Pigeon.Result<Void>) {
        scanner.scan()
        result.success(null)
    }
}
```

Listing 4.6: Android plugin main class.

Actual scanning is encapsulated in the scanner classes - *BluetoothScanner* for Bluetooth 4 & 5 and *WifiNanScanner* and *WifiBeaconScanner* for Wi-Fi technologies.

Wi-Fi scanners are split because there are separate packages for the Wi-Fi Beacon and Wi-Fi NaN on the Android platform. First, we will examine the *WifiScanner* class, reading Wi-Fi Beacons. We can use the Beacon frames to identify aircraft as if it was a network we are trying to connect to. At the start of scanning, we register the receiver object, which will handle receiving

beacon packets. This object has a method that is called every time a new advertisement is received. Then we need to parse the data using *OdidMessageHandler* and send them to the correct message stream, according to the message type. For each scan entry, we conveniently get an instance of *ScanResult*. Apart from advertisement data, this class holds information about the device broadcasting it that will be useful for us, namely the MAC address and Received Signal Strength Indication (RSSI).

```
fun scan() {
  context.registerReceiver(broadcastReceiver, IntentFilter(
       WifiManager.SCAN_RESULTS_AVAILABLE_ACTION))
  val ret = wifiManager!!.startScan()
}
private val broadcastReceiver = object : BroadcastReceiver() {
  override fun onReceive(contxt: Context?, intent: Intent?) {
    for (scanResult in wifiList) {
      handleResult(scanResult)
    }
  }
  fun handleResult(scanResult: ScanResult) {
       for (element in scanResult.informationElements) {
         if (element != null && element.id == 221) {
              val buf: ByteBuffer = element.bytes
              processRemoteId(scanResult, buf)
          }
       }
  }
 fun processRemoteIdscanResult: ScanResult, buf: ByteBuffer) {
       val arr = ByteArray(buf.remaining())
       if (isRemoteIdMessage())
       {
           val byteBuffer = ByteBuffer.wrap(arr, 1, 25)
           byteBuffer.order(ByteOrder.LITTLE_ENDIAN)
           val type
             = Pigeon.MessageType
             .values()[messageHandler.
               determineMessageType(arr, 1)]
           if(type == Pigeon.MessageType.BasicId)
           {
               val message: Pigeon.BasicIdMessage?
                              = messageHandler.fromBufferBasic(
                              arr, 6, scanResult.BSSID)
               message?.source = Pigeon.MessageSource.WifiBeacon;
               message?.rssi = scanResult.level.toLong();
               basicMessagesHandler.send(message?.toMap() as Any)
           }
           else if(type == Pigeon.MessageType.Location)
           // ...
       }
   }
```

Listing 4.7: Implementation of reading Wi-Fi Beacons.

In the case of *WiFiNaNScanner*, we have a very similar concept of setting a listener, parsing data, and sending them to streams, just with one exception, which is that we first need to check whether the given device even supports NaN technology. The class uses *WifiAwareManager*. Firstly, we request a session creation and register the receiver to see whether the attempt was successful. If yes, we can be sure that the technology is supported and start actual scanning. We save the created session and subscribe to *ServiceDiscovered* event.

Contrary to the beacons, results from these scans do not include MAC address or RSSI. In the documentation, creators write that "aware discovery does not provide the MAC address of the peer" [55]. This can be a big complication because we group the message packs according to the MAC address. If the device uses NaN and some other standard, we will not be able to recognize that pack comes from the same device.

```
fun scan() {
  private val myReceiver: BroadcastReceiver =
        object : BroadcastReceiver() {
        override fun onReceive(context: Context?, intent: Intent?) {
          if (wifiAwareManager!!.isAvailable) {
                startScan()
          }
        }
        // check support on device
        if (!context.getPackageManager().hasSystemFeature(
          PackageManager.FEATURE_WIFI_AWARE)) {
            return;
        }
        context.registerReceiver(myReceiver, IntentFilter(
          WifiAwareManager.ACTION_WIFI_AWARE_STATE_CHANGED))
    }
    private val attachCallback: AttachCallback =
object : AttachCallback() {
        override fun onAttached(session: WifiAwareSession) {
            wifiAwareSession = session
            wifiAwareSession!!.
              subscribe(config, object  DiscoverySessionCallback() {
                override fun onServiceDiscovered(
                    peerHandle: PeerHandle?,
                    serviceSpecificInfo: ByteArray?,
                    matchFilter: MutableList<ByteArray>?){
                       receiveDataNaN( serviceSpecificInfo,
                         peerHandle.hashCode(), timeNano,
                         transportType)
                   }
                })
             }
        }
```

Listing 4.8: Implementation of reading Wi-Fi NaN advertisements (1st part).

```
    fun receiveDataNaN(
      data: ByteArray?, peerHash: Int, timeNano: Long,
      transportType: String?
    ) {
      val byteBuffer = ByteBuffer.wrap(data, 4, 25)
      val type = Pigeon.MessageType.
        values()[messageHandler.determineMessageType(data, 4)]
      // rest is the same as in previous code snippet
      })
```

Listing 4.9: Implementation of reading Wi-Fi NaN advertisements (2nd part).

Lastly, we will take a look at the *BluetoothScanner* class, handling both Bluetooth 4 & 5 advertisements.

```
fun scan() {
        var scanSettings = ScanSettings.Builder()
                .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
                .build()
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O &&
                bluetoothAdapter.isLeCodedPhySupported &&
                bluetoothAdapter.isLeExtendedAdvertisingSupported
        ) {
            scanSettings = ScanSettings.Builder()
                    .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
                    .setLegacy(false)
                    .setPhy(ScanSettings.PHY_LE_ALL_SUPPORTED)
                    .build()
        }
        bluetoothAdapter.bluetoothLeScanner.startScan(scanFilters,
          scanSettings, scanCallback)
    }
private val scanCallback: ScanCallback = object: ScanCallback(){
        override fun onScanResult(callbackType: Int,
                        result: ScanResult) {
            val scanRecord: ScanRecord = result.scanRecord ?: return
            val bytes = scanRecord.bytes ?: return
            val type = Pigeon.MessageType.
              values()[messageHandler.determineMessageType(bytes, 6)]
            if(type == Pigeon.MessageType.BasicId)
            {
                val message: Pigeon.BasicIdMessage? = messageHandler
                  .fromBufferBasic(bytes, 6, result.device.address)
                message?.source
                  = Pigeon.MessageSource.BluetoothLegacy;
                message?.rssi = result.rssi.toLong();
                basicMessagesHandler.send(message?.toMap() as Any)
            }
            else if(...) //  other message types follow
        }
  }
```

Listing 4.10: *BluetoothScanner* class methods.

Class uses *BluetoothAdapter* from the *android.bluetooth package*. The adapter has knowledge about which standard is available and provides the *startScan* method, which takes a callback as an argument. So again, we register our receiver and parse data using our parser. The situation is very similar to the *WifiScanner*, we also get instances of *ScanResult* when our *scanCallback* is called.

4.1.6 iOS Native Code

For the iOS plugin, we used the Swift language together with Objective-C. Our model classes generated by Pigeon are in Objective-C, but this is not a challenge since we can use them also in Swift. For writing our code, we will use just Swift. We also need the main plugin class. In this case, it is the class *SwiftFlutterOpendroneidPlugin*. Similarly to the Android platform, we need to implement the *FlutterPlugin* interface and the *HostAPI*. In the main class, we need to set up the plugin by connecting to *EventChannels* and initializing the scanners. These tasks are completed within the *register* method.

```
public class SwiftFlutterOpendroneidPlugin
  : NSObject, FlutterPlugin, DTGApi{
    private var bluetoothScanner: BluetoothScanner? = nil
    private let locationMessagesStreamHandler = StreamHandler()
    //..
    public static func register(
        with registrar: FlutterPluginRegistrar) {
        let messenger: FlutterBinaryMessenger =registrar.messenger()
        let instance :
          SwiftFlutterOpendroneidPlugin & DTGApi & NSObjectProtocol
            = SwiftFlutterOpendroneidPlugin.init()
        DTGApiSetup(messenger, instance);
        // Event channels for every message type
        FlutterEventChannel(name: "flutter_location_messages",
          binaryMessenger: registrar.messenger())
           .setStreamHandler(instance.locationMessagesStreamHandler)
        //...
        instance.bluetoothScanner = BluetoothScanner(
            locationMessageHandler:
                instance.locationMessagesStreamHandler,
        )
    }
}
```

Listing 4.11: Part of Implementation of Swift Plugin.

As found out during the analysis stage, the only technology we are able to implement is the Bluetooth 4. This task is realized in a class *BluetoothScanner*. We take advantage of *CoreBluetooth*'s *CBCentralManager* objects. Such objects manage discovered or connected remote peripheral devices, including scanning for, discovering, and connecting to advertising peripherals. Peripherals are represented by *CBPeripheral* objects[47].

When initializing the scanner, we instantiate the *CBCentralManager*. It requires a so-called delegate, which is an object that will receive events from the *CBCentralManager*. In this case, the delegate is our *BluetoothScanner*. Such a delegate has to implement *CBCentralManagerDelegate* protocol. A protocol is just a term that Apple uses instead of an interface. A method that is crucial to implement and will be called on our delegate on a new device discovered is the *centralManager(CBCentralManager, didDiscover: CBPeripheral, advertisementData: [String : Any], rssi: NSNumber)* method, which conveniently passes the peripheral object, advertisement data, and signal strength indication.

```
class BluetoothScanner: NSObject, CBCentralManagerDelegate {
    var centralManager: CBCentralManager
    func scan() {
        centralManager.scanForPeripherals( withServices: nil,
            options: [
                CBCentralManagerScanOptionAllowDuplicatesKey: true,
            ]
        )
    }
    func centralManager(_ central: CBCentralManager,
      didDiscover peripheral: CBPeripheral,
      advertisementData: [String : Any], r
      ssi RSSI: NSNumber) {
        guard let data = getOdidPayload(advertisementData) else {
            // This advertisement is not an ODID ad data
            return
        }

        do {
            var err: FlutterError?
            let typeOrdinal = UInt(exactly: dataParser.
                determineMessageTypePayload(data, offset: 6,
                error: &err)!)
            let type = DTGMessageType(rawValue: typeOrdinal!)
            if(type == DTGMessageType.basicId)
            {
                let message : DTGBasicIdMessage? =
                        dataParser.fromBufferBasicPayload(data,
                  offset: 6,
                  macAddress: peripheral.identifier.uuidString,
                  error: &err)
                message!.rssi = RSSI.intValue as NSNumber
                basicMessageHandler.send(message!.toMap() as Any)
            }
            else if(type == DTGMessageType.location)
            ..//
            }
        }
```

Listing 4.12: Implementation of *BluetoothScanner* in Swift(1st part).

```
    func centralManagerDidUpdateState(_ central: CBCentralManager) {
        stateHandler.send(central.state.rawValue)
        updateScanState()

        if (central.state == .poweredOn && autoRestart) {
            scan()
        }
    }
}
```

Listing 4.13: Implementation of *BluetoothScanner* in Swift (2nd part).

4.2 Example Application Implementation

The final part of the project is the example receiver implementation for OpenDroneID Bluetooth, Wi-Fi NaN and Wi-Fi Beacon signals for Android and iOS phones. The application is compliant with the Bluetooth, Wi-Fi NaN, and Wi-Fi Beacon parts of the *ASTM F3411* Remote ID standard and the *ASD-STAN prEN 4709-002* DRI standard, described in the research chapter.

The application continuously scans for Bluetooth advertising, Wi-Fi Beacon frames and Wi-Fi NaN signals. Suppose any is found matching the specifiers for remote ID signals. In that case, it adds that device to a list and displays the drone's location on a map. Moreover, the detailed view of an aircraft will show the exact content of the remote ID data.

4.2.1 Project structure

To maintain a clean codebase, source code is separated into classes grouped in several directories. At the root of the project, we have the *main.dart* file and *app.dart* with root application widget. The main file instantiates the root application widget and thus starts the application. The following list describes the project structure.

- Widgets - definitions of UI components, further structured according to widgets concern - map, toolbar, sliders widgets.
- BLoC - state-management classes that use *Bloc* package.
- Utils - helper functions for CSV logging or reading JavaScript Object Notation (JSON) files
- Constants - immutable parameters, such as application styling

4.2.2 State Management

For each concern in the app, we will create one cubit. Firstly, the *OpenDroneIDCubit* will manage the connection to the library and listen to streams providing message packs. Secondly, we will have cubits handling the data inside

the application - *AircraftCubit, ZonesCubit, StandardsCubit, SelectedItemCubit* and cubits managing states of the widgets - *MapCubit, SlidersCubit*. Each cubit has its state object. If the state changes, the cubit emits a message to widgets that want to watch for changes in the state, and those widgets then rebuild themselves.

```
Text(
  context.watch<AircraftCubit>().state.packs.length.toString()
                + " Drones Around"),
),
```

Listing 4.14: Setting listener to number of detected drones.

We have one instance of each cubit available in the whole application widget tree. We instantiate and register the cubits in the main file. If we want to invoke method on a cubit, we use *context.read<CubitName>().doSomething*. It is important to use *read* here instead of the *watch* because we do not want to rebuild the caller widget when we are invoking a method on a cubit.

```
class PlayButton extends StatelessWidget {
  const PlayButton({
    Key? key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final scanningActive =
        context.watch<OpendroneIdCubit>().isScanningBluetooth ||
            context.watch<OpendroneIdCubit>().isScanningWifi;
    return InkWell(
        onTap: () {
          scanningActive
              ? context.read<OpendroneIdCubit>().stop()
              : context.read<OpendroneIdCubit>().start();
        },
        child: ...
      ),
    );
  }
}
```

Listing 4.15: Reading the cubit from build context and invoking method.

Another useful way to listen to changes in the cubit state is to use the *BlocBuilder*. It is used when we want to draw a Widget based on what is the current state, and the widget depends on one cubit. In the *build* method, we wrap the widget with *BlocBuilder* and specify which cubit we will use. Then, in the builder method, we get an instance of the state as a parameter and can create UI components accordingly. This way, we can use just a stateless widget, and still, the UI will be rebuilt when the state changes. In the following code snippet, we can see an example of *BlocBuilder* usage. Note that, for instance, the button fill color or icon is assigned according to the state.

```
class ScanningStateIcons extends StatelessWidget {
  const ScanningStateIcons({
    Key? key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final theme = Theme.of(context);
    return BlocBuilder<OpendroneIdCubit, ScanningState>(
      builder: (context, state) {
        return Row(
          children: [
            RawMaterialButton(
              onPressed: () {
                if(state.usedTechnologies ==
                                UsedTechnologies.Bluetooth){
                  context.read<OpendroneIdCubit>().setBtUsed(false);
                } else {
                  context.read<OpendroneIdCubit>().setBtUsed(true);
                }
              },
              elevation: 2.0,
              padding: const EdgeInsets.all(8),
              constraints: const BoxConstraints(minWidth: 0),
              fillColor: state.isScanningBluetooth
                  ? theme.colorScheme.primary
                  : theme.colorScheme.background,
              child: Icon(
                state.usedTechnologies == UsedTechnologies.Bluetooth
                    ? Icons.bluetooth
                    : Icons.bluetooth_disabled,
                color: Colors.black,
                size: 25,
              ),
              shape: const CircleBorder(),
            ),
          ],
        );
      },
    );
  }
}
```

Listing 4.16: *BlocBuilder* example.

4.2.3 Using Own Library

At the startup of the scanning for nearby devices, we need to set up a listener to the message packs stream provided by the library. Every time we receive the data from the library, *scanCallback* is invoked. The following snippet shows the *OpenDroneIdCubit* and its state and setting up the listener when the *start* method is called. The cubit also has methods to start and stop scans, so if we

create a button to start scans, we do not have to pass a callback as a parameter. We can get the cubit from the build context, as shown in listing 4.12.

```
 class ScanningState {
  bool isScanningWifi;
  bool isScanningBluetooth;
  UsedTechnologies usedTechnologies;

  ScanningState({
    required this.isScanningWifi,
    required this.isScanningBluetooth,
    required this.usedTechnologies,
  });
}

class OpendroneIdCubit extends Cubit<ScanningState> {
  StreamSubscription? listener;
  AircraftCubit aircraftCubit;
  OpendroneIdCubit(
      {required this.mapCubit,
      required this.selectedAircraftCubit,
      required this.aircraftCubit})
      : super(ScanningState(
          isScanningBluetooth: false,
          isScanningWifi: false,
          usedTechnologies: UsedTechnologies.Both,
        ));

  void scanCallback(MessagePack pack) {
    aircraftCubit.addPack(pack);
   }

  Future<void> start() async {
    listener = FlutterOpenDroneId.allMessages.listen(scanCallback);
    await FlutterOpenDroneId.startScan(state.usedTechnologies);
    updateScanningStateBluetooth();
    updateScanningStateWifi();
  }
```

Listing 4.17: Setting listener and starting scan.

4.2.4 Implementing Graphical Interface

When implementing GUI applications using the Flutter SDK, developers should obey specific rules and best practices. Apart from effective state management, it is also essential to split widgets into smaller ones and delegate building parts of the widget to build methods. Properties should not need to be propagated to children's widgets more than one or two layers deep. If this happens, or more widgets use the same data, we will create a cubit.

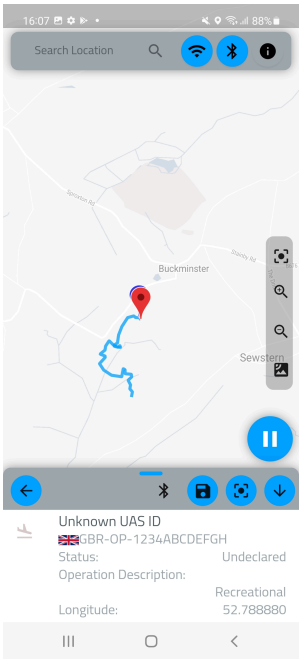

Figure 4.1: Main Page

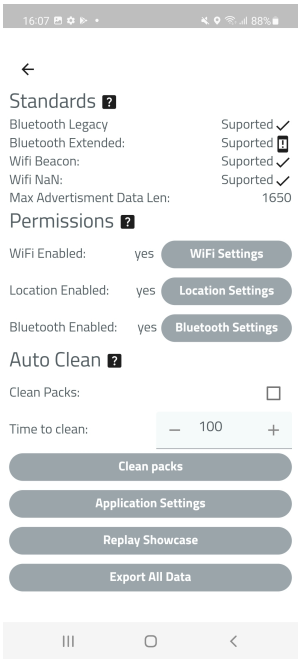

Figure 4.2: About Page

The GUI will follow the guidelines on Material Design. Material is an adaptable system of guidelines, components, and tools that support the best practices of UI design [56]. We will utilize the Flutter Material library, which contains Flutter widgets implementing Material Design [57]. As a first step, we need to implement the basic structure and layout of the application. For this purpose, we can use the *Scaffold* widget. Then, our design will consist of one main screen and a secondary settings screen. Widgets that cover the whole screen are called pages. Our main page will include the map underlay, a map toolbar with search and map settings, and a panel with a list of detected devices or detail with information about the particular device. Screenshots of application GUI are displayed on figures 4.1 and 4.2

All the information about detected aircraft will be presented on a separate widget, which will slide up from the bottom of the screen. If the user selects one device from the map, all the information will be shown on this widget. Otherwise, this widget will contain a scrollable list of all the devices. To achieve the sliding-up behavior, we can once again use a library widget called *SlidingUpPanel*. Based on the Material Design bottom sheet component, this widget works on both Android & iOS [58]. Users will also be able to hide the panel, so the map covers the maximum space available. The slider works in two modes. It either shows a complete list of detected aircraft or zones, with a possibility to filter results. Secondly, after specific aircraft is selected from the map, the slider will contain detailed message contents.

Figure 4.3 shows an application running on iPhone 8 with a significantly smaller screen. Positions and sizes of widgets are defined relatively to screen size, so the ratio of sizes of components remains the same on all screen sizes. We used the *MediaQuery.of(context).size* to find out the screen size of a device running the application. For example, the minimum height of the slider is defined as one-fourth of the screen height.

Some features are not available on iOS, therefore widgets containing them are now shown. For instance, the Wi-Fi status icon on the main toolbar is present just on Android. To find out which system we are running and other information about the environment, we used the *Platform* [59] class, specifically the *Platform.isAndroid* property.

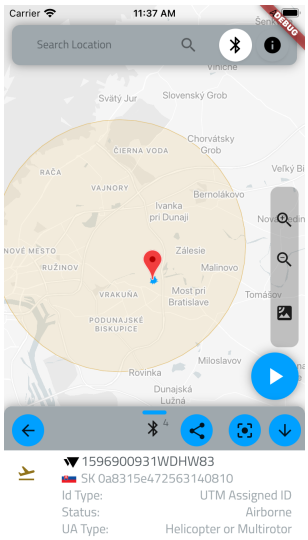

Figure 4.3: Application Running on iPhone 8

Finally, there will be the *"About"* page, with information about used standards, permissions, and other settings. Users can see what standards are supported on their devices with a short explanation on this page. We also placed buttons that open system settings, enabling users to turn on certain permissions quickly. Lastly, there is an option to automatically remove message packs from the device, from which the last pack was received a certain optional time ago.

The following figure 4.4 shows the internal structure of the graphical interface and business components. The widget tree was not modeled to every detail, some parts from the bottom layers were omitted to keep the diagram readable. Apart from the widget tree, we can see the business components and their relations with widgets.

A green arrow from the cubit to the widget means, that widget watches cubits states and rebuilds accordingly. The blue arrow symbolizes method calls on cubits.

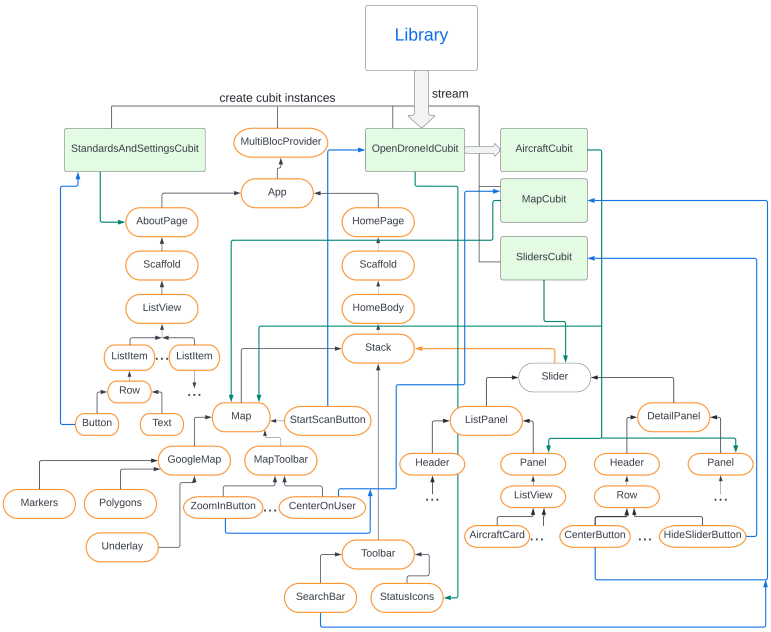

Figure 4.4: Simplified Widget Tree

Map

For the map underlay, we first have used the Flutter Map package [60], with map data provided by OpenStreetView maps. This decision turned out to be not ideal during the implementation, as these maps had fewer customization options and lacked the efficiency needed to run smoothly. So we have decided to use Google Maps Flutter package [61]. The package is widely used and thus supposedly well maintained, and the significant advantage is that users are accustomed to using the maps. It also allows using comprehensive styling and customizations. To represent detected aircraft, we will use *Marker*, generated from our list inside *Aircraft Cubit*, to show the trajectory, we will use *Polyline* and finally, for zones, we will use *Polygon* or circle, according to zone shape. The zone shape will be filled with different colors for different zone types. These components will respond to taps, highlighting the aircraft or zone and showing its details.

We want to be able to change the map programmatically. For example, when a user taps the button that centers the map on his location. We can achieve this by storing the instance of *GoogleMapController* inside the *MapCu-*

bit. We encapsulate map states and methods for controlling the map inside this cubit, which will be available for every widget in the build context. Methods to manipulate the map will simply use the interface of the controller. The *MapCubit* will contain all methods for setting map style, zoom and placing pins to the map.

```
Future<void>? centerToUser() {
    controller?.getZoomLevel().then((currentZoomLevel) {
      moveCamera(
        gmap.CameraUpdate.newCameraPosition(
          gmap.CameraPosition(
            target: state.userLocation,
            zoom: currentZoomLevel,
          ),
        ),
      );
    });
  }
  Future<void>? centerToLoc(gmap.LatLng loc) {
    controller?.getZoomLevel().then((currentZoomLevel) {
      moveCamera(gmap.CameraUpdate.newCameraPosition(
          gmap.CameraPosition(target: loc, zoom: currentZoomLevel)))
    });
  }
```

Listing 4.18: Centering map to location.

In a landscape mode, the layout of the main page changes. Since we have less vertical space, the slider can only be hidden or maximized. The button to start scans moves to the left side of the screen because on the right, the map toolbar takes all the available vertical space. Concerned widgets use the *MediaQuery.of(context).orientation* to find out how the phone is orientated in their *build* methods. Figure 4.5 shows the application in landscape mode.

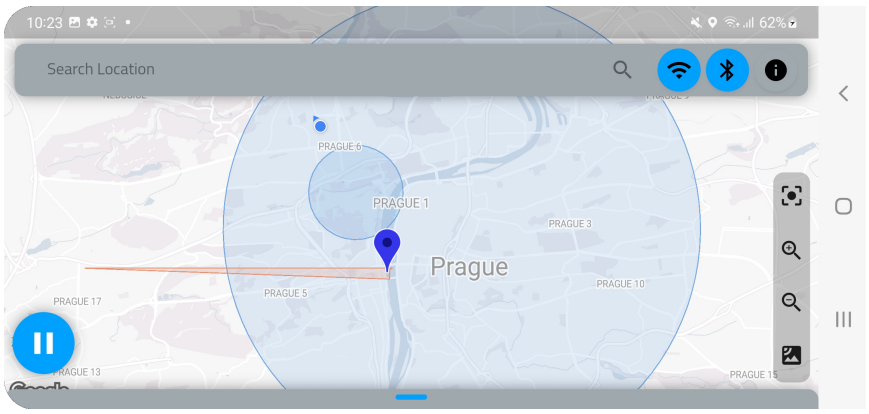

Figure 4.5: Application in a Landscape Mode

For the map search, we will use API provided by Google Maps. Imple-

mentation of the search was inspired by the article on the Flutter Campus learning site [62]. The API gives us tools to create an easy-to-use search with suggestions. After the user inputs the text, we call the method to retrieve location data and add a pin to our map. Users will benefit from the possibility of searching for locations if they do not want the application to access their location or want to see the traffic in a specific area.

4.2.5 Exporting Messages in a CSV Format

When implementing this functionality, we need to fulfill two tasks - create a CSV file and then save it. For creating a CSV file, we will use package called *csv 5.0.1* [63]. We will use the package to convert a list of rows where every row is a list of values to CSV string, which we can then save to a file.

The original idea was to export the file directly to device storage, such as the Downloads directory. The user would pick the desired export location with a file picker. Unfortunately, we were just able to find ways to save files to temporary directories or application documents directory, but not to common storage space. Application documents directory is a directory for the application to store files that only it can access, therefore not useful for us. The system clears the directory when the application is deleted [64]. There exists a package *file_picker* [65] with the *saveFile* method that allows users to pick the save path, but the method is not available for iOS nor for Android.

Consequently, we have decided to use a Flutter plugin *share_plus* [66] to share content from the application via the platform's share dialog. This way, the user can choose what application to share the CSV file with. As a compromise, we also implemented saving the file directly to the downloads folder, but this approach works just on Android. After the file is saved, we inform the user with a simple info dialog.

```
// aircraft_panel.dart - invocation
context.read<AircraftCubit>().
        exportPackToCSV(messagePackList.last.macAddress, true)
        .then((value) => showInfoDialog(
                context, "Saved successfuly to " + value));

// saving file to downloads
 Directory generalDownloadDir
   = Directory('/storage/emulated/0/Download');
 final pathOfTheFileToWrite =
        generalDownloadDir.path + "/csv_export$name.csv";
 File file = File(pathOfTheFileToWrite);
 file = await file.writeAsString(csv);

// opening native sharing menu
Share.shareFiles([pathOfTheFileToWrite], text: 'Your Data');
```

Listing 4.19: Creating and saving a CSV file.

4.2.6 Application Tutorial

For almost all the applications published to the application stores, the tutorial is one of the essential things the developers should worry about. If the application feels overwhelming and complicated to the users, they probably will not want to use it. We need to explain the workflow of the application simply and effectively. For this purpose, we chose the *Showcase* package [67]. The basic idea is to guide users through our widgets by highlighting them and presenting text explaining the widget concept and usage. This way, we go step by step through the whole application. The showcase will be shown after the first startup but can also be replayed later.

In the figure 4.6, we can see an example of a showcase of the aircraft list.

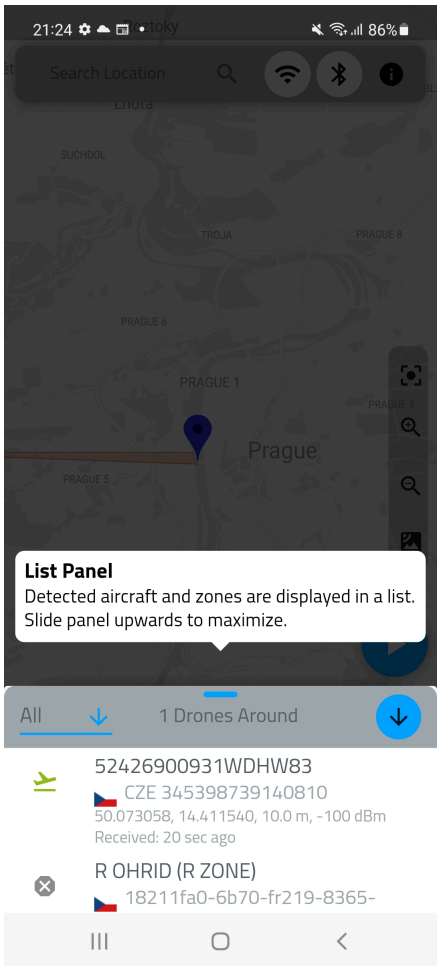

Figure 4.6: Showcase with a Description of Aircraft List

To add a widget to showcase, we need to wrap it with the *Showcase*, provide the key and the text that will be presented to the user. We store keys and texts in a *ShowcaseCubit*. We also need to wrap our pages with *ShowCaseView* widget, that has a method *startShowcase*. We will pass a list of our keys to this method and start the showcase.

```
Showcase(
        key: context.read<ShowcaseCubit>().droneDetailLockKey,
        description: context.read<ShowcaseCubit>().
                droneDetailLockDescription,
        title: "Aircraft Detail",
        child: Container(....),
        )
```

Listing 4.20: Wrapping a container with a *Showcase*.

4.2.7 Styling

Flutter provides the concept of *Themes* for styling the application. *Theme* is a class that defines the styling for the application, setting colors, fonts, and all the other decorations either for the whole application or for specific widgets, like text inputs. The theme is then assigned to the *MaterialApp* widget and can be retrieved from the *BuildContext* by calling *Theme.of(context)* in the place where we want to access the theme object. After loading the theme, stylings for the widgets are applied automatically. The Dronetag application inspires a particular theme we used for this application. Same fonts and primary and secondary colors were applied.

4.2.8 Map Styling

Google Maps Flutter package allows us to define a style for the map, and we can, for example, change the colors of map features, like buildings or roads. Once again, we used the style from the Dronetag application, which is stored in a JSON file. This file is then loaded using utility class *GoogleMapsStyleReader* and applied after the startup or when a user changes the type of the map. Of course, style is not applied to the satellite map.

4.2.9 Releasing the Application

To publish the application to the store, we need to create a release build, which means a minimum size build with optimizations. Furthermore, we will provide an application icon and code signature and complete an application bundle used for distribution. When releasing for the Android platform, we will follow the steps from the article from Flutter documentation [68]. For the iOS platform, we can also draw from the article [69].

For icons, we can simplify the process with the package *LauncherIcons* [70]. It is a command-line tool that simplifies the task of updating the icon.

We just add an icon to our *assets* directory, and the tool with *flutter pub run flutter_launcher_icons:main*. We used an icon downloaded from *Freepik* portal [71], that is free for commercial purposes. To help with creating an application bundle for Android and Google Play, we will use the *bundletool* [72]. It is a command-line tool used to build an Android App Bundle, and convert a bundle into the various Android Application Package (APK)s that are deployed to devices. The APK can be then installed onto a phone so we can test it with users before uploading it to a store.

Chapter 5

Testing

The solution needs to be tested on various levels. In the first place, we have to test the library, mainly receiving the data using all the supported standards. To simulate ongoing traffic, we can use the transmitter code for Raspberry Pi, created by a Soren Fris [73]. The program supports transmitting static drone ID data via Wi-Fi Beacon or Bluetooth. It sends just static data since the primary purpose is to demonstrate how to set up the transmission.

The transmitter has its limitations, given that neither Raspberry nor the transmitter program does support the Wifi NaN standard. During the process of implementation, we acquired an ESP32-C3-DevKitM-1 [74], which is a system on a chip that integrates the Wi-Fi (2.4 GHz band) and Bluetooth features. Using this kit, we can try Wi-Fi NaN and Bluetooth 5 transmission. Finally, we had an opportunity to test the application with Dronetag Mini devices. These devices are mounted on a drone and broadcast drone info with Bluetooth. They already fulfill all the new regulation necessities, so it was a great way to try our application with a piece of equipment that will be used in the real world.

An important part of the process is using suitable smartphones because, as explained in the research chapter, they may support a different combination of technologies. We cannot use simulators running on desktop computers because such simulators do not have access to computer peripherals. During the implementation, we used the iPhone XR supporting just the Bluetooth Legacy to test running the application in the iOS environment. For the Android platform, we had the Huawei Mate 9 with support of Bluetooth 4 and WiFi Beacon, and finally, the Samsung A52S, which supports all the standards we need.

Secondly, we want to access the usability of the example application. This is done best when testing with actual users. We organized a test session with volunteers representing various groups of future users. We gave them a few tasks to complete and also let them explore the application. After that, we collected their feedback and suggestions. From these results and heuristical

analysis of the GUI, we concluded several issues and assigned them a priority for future development. We also used code analysis to detect technical deficiencies in the code, as well as performance assessment tools to track down inefficiencies in the widget tree.

5.1 User Testing

In total, we had six testers. We could split them into three groups. The first group was made up of 2 employees of the Dronetag company. We may consider them expert users because they are already familiar with the drone problematics and concepts of Remote ID. Will will mark this group as *Group A*. Secondly, we have a group of bachelor's students in information technology, the *Group B*. They are not as competent in UAV industry, but they study informatics and develop software themselves. The final *Group C* consists of members of the general public with no background in software development or UAV industry.

5.1.1 Test Scenarios

The scenario for testing is straightforward. In the beginning, we present the application tutorial to the test subjects. After browsing through the tutorial showcase, we gave them a few tasks to complete.

First, we want users to identificate the aircraft that is the closest to them. In the second task, we give the user a specific location to find and see the traffic there, check whether certain aircraft infiltrates forbidden zones, and identify the operator of such aircraft. We then encourage them to export gathered data from one aircraft to the application of their preference.

5.1.2 Testing Evaluation

In general, we may say that the application received positive feedback. There was no problem comprehending the fundamental ideas used when designing the UI. Users liked the simplicity of the layout and placement of the widgets and styling. On the other hand, several minor problems confused users, mainly from *Group C*. *Group A* also introduced several possible improvements to application functionality. The following paragraphs will go through the obstacles testers encountered when completing the test scenario.

Right from the start, 3 of 6 testers reported difficulties with the tutorial. They did not immediately know how to skip the showcase or transfer to the next item in a showcase, even though the instructions were written in the first showcase text. The instructions were not clearly visible or highlighted. In addition, the showcase is probably too long, and users are not able to recall all the information presented to them at the start. This was proved true in

2 cases when testers, after seeing the tutorial, did not recall which button is used to start scans, which is shown at the beginning of the tutorial.

The next problematic spot is the handling of a panel that slides up. Users can use the panel in collapsed or maximized form or hide it with a button. Collapsing or maximizing the panel is done by sliding the panel up or down. Testers tended to slide down the panel even if it was collapsed, as they expected it would hide it. The direction in which they can manipulate the slider was not shown.

Experienced users also came up with improvement ideas that could be done to the map. In the current state, the drone's direction is not shown on the map. Filter that enabled users to see just drones or just zones is not applied to the items on the map, just to the list. One tester also suggested that the aircraft should be sorted according to distance from the user automatically and by default. If we have more drones around that are constantly broadcasting, sorting by time can be problematic as the order is changed frequently, and users may perceive the frequent re-sorting as chaotic.

Lastly, testers pointed out that after data were shared, for example, with an email application, there was no confirmation whether the action was successful.

5.2 Heuristic Analysis

To analyze created UI, we will perform an analysis according to Jakob Nielsen's ten general principles for interaction design. They are called "heuristics" because they are broad rules of thumb and not specific usability guidelines [75].

- *Visibility of system status*

  The design should always keep users informed about what is going on through appropriate feedback.

  Scanning state and active technology are visible to the user on icons. After exporting data, users get feedback about how the saving is finished. After the user decides to share data, there is no further notification about how the operation ended. For example, when an email is sent, there is no confirmation that it was sent successfully.

- *Match between system and the real world*

  The design should speak the users' language and follow real-world conventions and concepts familiar to the user.

  We have used Google Map for map underlay. The maps from Google are commonly used in many applications. Therefore, we conclude that its concepts are familiar to users. Sliding-up panel is also commonly used in applications with maps.

- *User control and freedom*

  Users often perform actions by mistake. They need a clearly marked "emergency exit" to leave the unwanted action.

  There are back buttons in two places: from the aircraft detail back to the list or from the settings page to the map page. There are not any more actions that could be done by mistake.

- *Consistency and standards*

  Follow platform and industry conventions.

  Apart from using a standard map, the whole application also followed technical standards by ASD-STAN, which are meant for an entire UAV industry. Platform conventions were fulfilled by using *Material* design components.

- *Error prevention*

  The best designs carefully prevent problems from occurring in the first place.

  The only destructive action in the application is deleting received message packs. This action is done with a button click, without confirmation from the user.

- *Recognition rather than recall*

  Minimize the user's memory load by making elements, actions, and options visible.

  All important actions are visible from the main page. The only hidden functionality is the automatic deletion of packs after a certain time.

- *Flexibility and efficiency of use*

  Allow users to tailor frequent actions.

  The application does not have an alternative way of completing tasks since it is small in size. Also. the form factor of a mobile phone does not allow for shortcuts.

- *Aesthetic and minimalist design*

  Interfaces should not contain information that is irrelevant or rarely needed.

  Important information about aircraft is presented on a card in the list. Arguably, some data fields may not be comprehensible to some users, such as zone ID number. These should be hidden from cards in a list and shown just on the detail panel.

- *Help users recognize, diagnose, and recover from errors*

  Error messages should precisely indicate the problem and constructively suggest a solution.

  We identified one error state that users can experience - when a map or location search is used without a connection to the internet. Now, the user is not notified about missing internet connection, map tiles are cached, but the search does not work.

- *Help and documentation*

  Provide documentation to help users understand how to complete their tasks.

  The tutorial showcase takes the users through the whole workflow of the application.

5.3 Code Analysis

Flutter provides some valuable tools to analyze created applications. We will use mainly the tools integrated into the *Android Studio* Integrated Development Environment (IDE). Firstly, we will run the static code analysis with the option *Inspect code*. The tool will notify us about unused declarations and variables that could be constants and many more improvements.

Next, we will use the Flutter Android Studio plugin, which enables us to show performance data. The tool will show us which widgets are rebuilt and when, so we can identify unnecessary rebuilds. We have found out that the map widget is rebuilt frequently after getting every new message. This can cause performance concerns. The issue should be addressed by splitting the widget into smaller widgets and ensuring that the map features such as Polygons and Markers are not repainted when they do not need to.

5.4 Identified Issues

After completing code analysis user tests and heuristical analysis of the application, we identified a list of issues that needs to be solved. We assigned a priority to every issue and outlined possible resolutions. The table 5.1 sums up the issues.

Issue	Description	Solution	Priority
1	Showcase: unclear control	Add ”skip”, ”next” buttons, counter of showcase elements	8
2	Showcase: too long	Remove unnecessary showcases	4
3	Slider: unclear how to hide	Implement hiding slider with slide gesture	5
4	Filtering of map elements	Connect list filters to map widget	5
5	Completion of share action is not confirmed	Add text informing user about outcome of share action	3
6	Deletion of data is not confirmed	Add confirmation dialog	5
7	Sorting according to time causes list to redraw frequently	Add delay between receiving message and sorting the data	4
8	Unnecessary information on list card	Remove some fields from card	2
9	User is not notified, when internet is down	Add timeout, after which, user is notified	5
9	Unnecessary widget, rebuilds	Split the map widget into smaller pieces	7
10	No MAC address, in NaN scan results	Group messages according to other parameter	9

Table 5.1: GUI Problems Summary

5.5 Future Improvements

Regarding future improvements, there are two main ways to improve the solution. The first one is to improve the backend library. We expect that newer standards will be used more widely in the future. Specifically, we will wait for Apple to enable developers to access Wi-Fi and Bluetooth 5 scanning API. That would allow us to implement a solution equivalent to the Android one. In the current state, iPhone users can scan just using Bluetooth 4.0. Another issue that needs to be solved is that Wi-Fi NaN scanning does not provide the MAC address, so we cannot join it with messages from the same source but different technology.

Secondly, the example application could also be enhanced, as we have seen during the user testing. Significantly, the application tutorial should be improved, as it was confusing for some users. Buttons to manipulate the

showcase and counter that would show the position in a showcase are needed. There is also work to be done regarding the overall performance and efficiency.

The most significant setback regarding coding style and project structure is that we have a standalone stream for each message type, which we need to manage. Handling messages with polymorphism would be much more pleasant.

Chapter 6

Conclusion

In this thesis, we have successfully created a standalone library and example application for both leading mobile platforms. The solution is compatible with current regulations and standards of UAV industry. The system consists of a library implementing the scanning using wireless technologies and an example application presenting the data in a user-friendly way. During the development, we encountered certain problems on both levels that contributed to the solution's final state.

When designing the library, we found out that there was a lack of support for the technologies we wanted to utilize. The functionality, such as Wi-Fi scanning, needs to be written in platform-specific code. Therefore, we had different APIs on both platforms. Unfortunately, Apple does not allow 3rd party packages, and there is a lack of available APIs approved by Apple that would enable us to implement Wi-Fi or Bluetooth 5 scanning. This fact significantly altered the plan to create two equal applications, as on iOS, we were limited just to Bluetooth 4. In its current state, the application works best on Android phones.

The overall goal to publish the example application to the application stores was not completed, as the application is not yet in a production state. During testing, we gathered feedback from users, and several issues surfaced. These need to be addressed before releasing the application. We mainly used simulated drone transmissions to verify the communication. The application was released for further internal testing with Dronetag Mini devices, one of the first devices on the market. We were already able to test the solution in a real-world environment with these devices. It is a significant advantage because there are no other such devices available, and also, manufacturers still do not provide drones with Remote ID functionality.

The application development will continue in the future. We have identified usability problems that need to be addressed to improve user experience. Future Development also depends on whether Apple will enable certain functionality. There is also an opportunity to enroll in an Apple *MFi development*

program. Members are allowed to use otherwise inaccessible features. The program provides access to Apple proprietary technologies and components [76].To become a member, we must first submit the product plan and get approved.

In the EU, the regulations will be fully enforced next year. As we have seen in the existing applications section, there is no comparable solution to ours already available. Even though solutions for tracking drones exist, there is no versatile application on the market for scanning Remote ID from all UAV types. Thus, we still have enough time to overcome all the problems and ship the application to stores.

Bibliography

[1] Bělohlávková, V. Zásilky budou létat vzduchem. Doručování drony však komplikuje legislativa. [online], [cit. 2022-03-07]. Available from: https://www.idnes.cz/ekonomika/domaci/dron-zasilky-ukulele-praha-kytary.A220210_081042_ekonomika_vebe

[2] Bussiness Insider. Drone market outlook in 2022: industry growth trends, market stats and forecast. [online], [cit. 2022-03-06]. Available from: https://www.businessinsider.com/drone-industry-analysis-market-trends-growth-forecastsj

[3] Official Journal of the European Union. Commission Implementing Regulation (EU) 2021/664 of 22 April 2021 on a regulatory framework for the U-space (Text with EEA relevance) C/2021/2671. [online], [cit. 2022-03-06]. Available from: http://data.europa.eu/eli/reg_impl/2021/664/oj

[4] ASD-STAN. Introduction to the European UAS Digital Remote technical standard. [online], [cit. 2022-03-12]. Available from: https://asd-stan.org/wp-content/uploads/ASD-STAN_DRI_Introduction_to_the_European_digital_RID_UAS_Standard.pdf

[5] Official Journal of the European Union. COMMISSION DELEGATED REGULATION (EU) 2019/945. [online], [cit. 2022-04-25]. Available from: https://eur-lex.europa.eu/eli/reg_del/2019/945/2020-08-09

[6] Official Journal of the European Union. COMMISSION IMPLEMENTING REGULATION (EU) 2019/947. [online], [cit. 2022-04-25]. Available from: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32019R0947

[7] ASD-STAN. About ASD-STAN. [online], [cit. 2022-04-25]. Available from: https://asd-stan.org/about-asd-stan/

[8] ASD-STAN. ASD-STAN prEN 4709-002 P1. [online], [cit. 2022-04-24]. Available from: http://asd-stan.org/downloads/asd-stan-pren-4709-002-p1/

[9] Dronetag s.r.o. Remote ID Explained. [online], [cit. 2022-03-21]. Available from: https://help.dronetag.cz/knowledge-base/remote-id-explained/

[10] Poss, J and Zoldi, D.M.K. 3, 2, 1—Done! Remote ID Rule is Final. [online], [cit. 2022-03-06]. Available from: https://insideunmannedsystems.com/3-2-1-done-remote-id-rule-is-final/

[11] Official Journal of the European Union. COMMISSION IMPLEMENTING REGULATION (EU) 2022/425. [online], [cit. 2022-04-25]. Available from: https://eur-lex.europa.eu/eli/reg_impl/2022/425/

[12] Dronetag s.r.o. Remote identification device attachable to any drone. [online], [cit. 2022-03-21]. Available from: https://dronetag.cz/en/products/mini/

[13] Dronetag s.r.o. All-in-One Solution for Safe Drone Flights. [online], [cit. 2022-03-21]. Available from: https://dronetag.cz/en/product/

[14] Sattel, S. WiFi vs. Bluetooth: Wireless Electronics Basics. [online], [cit. 2022-03-07]. Available from: https://www.autodesk.com/products/eagle/blog/wifi-vs-bluetooth-wireless-electronics-basics/

[15] Bluetooth Special Interest Group. Vision and Mission. [online], [cit. 2022-04-17]. Available from: https://www.bluetooth.com/about-us/vision/

[16] GSM Arena. Flashback: a brief history of Bluetooth. [online], [cit. 2022-04-17]. Available from: https://www.gsmarena.com/flashback_a_brief_history_of_bluetooth-news-49119.php

[17] Øvrebekk, T. Bluetooth 5 Advertising Extensions. [online], [cit. 2022-05-02]. Available from: https://blog.nordicsemi.com/getconnected/bluetooth-5-advertising-extensions

[18] Wi-Fi Alliance. Who We Are - History. [online], [cit. 2022-04-17]. Available from: https://www.wi-fi.org/who-we-are/history

[19] Whitwam, R. Android O feature spotlight: Neighborhood Aware Networking (NAN) mode for WiFi. [online], [cit. 2022-03-21]. Available from: https://www.androidpolice.com/2017/03/21/android-o-feature-spotlight-neighborhood-aware-networking-nan-mode-wifi/

[20] Friis, S. Supported Smartphones. [online], [cit. 2022-03-15]. Available from: https://github.com/opendroneid/receiver-android/blob/master/supported-smartphones.md

[21] dlapilota.pl Sp. z o.o. Droneradar. [online], [cit. 2022-04-30]. Available from: https://play.google.com/store/apps/details?id=eu.droneradar.droneradar

[22] Unifly nv. Unifly launches e-Identification and tracking for drones. [online], [cit. 2022-04-30]. Available from: https://www.unifly.aero/news/unifly-launches-e-identification-and-tracking-for-drones

[23] Lougheed, P. Dart overview. [online], [cit. 2022-03-07]. Available from: https://dart.dev/overview

[24] Lougheed, P. A tour of the Dart language. [online], [cit. 2022-03-07]. Available from: https://dart.dev/guides/language/language-tour#important-concepts

[25] Oboh, A. Async Programming in Flutter With Streams. [online], [cit. 2022-04-18]. Available from: https://betterprogramming.pub/async-programming-in-flutter-with-streams-c949f74c9cf9

[26] Jovanoski, J. Reactive Programming in Flutter. [online], [cit. 2022-04-18]. Available from: https://betterprogramming.pub/reactive-programming-in-flutter-9fd7b0a4835/

[27] Amadeo, R. Google starts a push for cross-platform app development with Flutter SDK. [online], [cit. 2022-03-06]. Available from: https://arstechnica.com/gadgets/2018/02/google-starts-a-push-for-cross-platform-app-development-with-flutter-sdk/

[28] Flutter team. Flutter: the first UI platform designed for ambient computing. [online], [cit. 2022-03-06]. Available from: https://developers.googleblog.com/2019/12/flutter-ui-ambient-computing.html

[29] Flutter Team. The official package repository for Dart and Flutter apps. [online], [cit. 2022-03-23]. Available from: https://pub.dev

[30] Flutter Team. Flutter architectural overview. [online], [cit. 2022-03-11]. Available from: https://docs.flutter.dev/resources/architectural-overview

[31] Flutter Team. Introduction to widgets. [online], [cit. 2022-03-11]. Available from: https://docs.flutter.dev/development/ui/widgets-intro

[32] Flutter Team. Widget catalog. [online], [cit. 2022-03-11]. Available from: https://docs.flutter.dev/development/ui/widgets

[33] Bloc Community. Flutter_bloc 8.0.1. [online], [cit. 2022-04-15]. Available from: https://pub.dev/packages/flutter_bloc

[34] Kayfitz, B. Getting Started with Flutter BLoC Pattern. [online], [cit. 2022-04-18]. Available from: https://www.raywenderlich.com/4074597-getting-started-with-the-bloc-pattern

[35] Suri, S. Architect your Flutter project using BLOC pattern. [online], [cit. 2022-04-18]. Available from: https://medium.com/codechai/architecting-your-flutter-project-bd04e144a8f1

[36] Purwandaru, A. N. Getting Started with Flutter Bloc Pattern. [online], [cit. 2022-04-18]. Available from: https://www.mitrais.com/news-updates/getting-started-with-flutter-bloc-pattern/

[37] Geeks for Geeks. Flutter – Managing the MediaQuery Object. [online], [cit. 2022-04-14]. Available from: https://www.geeksforgeeks.org/flutter-managing-the-mediaquery-object/

[38] Baseflow. Permission Handler 9.2.0. [online], [cit. 2022-03-09]. Available from: https://pub.dev/packages/permission_handler

[39] Rawat, A. Creating a Flutter Plugin. [online], [cit. 2022-03-08]. Available from: https://medium.com/flutter-community/creating-a-flutter-plugin-dialog-box-78adbff15fe

[40] Sharma, A. How to develop a platform channel in Flutter between Dart and Native Code. [online], [cit. 2022-03-19]. Available from: https://medium.com/47billion/creating-a-bridge-in-flutter-between-dart-and-native-code-in-java-or-objectivec-5f80fd0cd713

[41] Pub.dev. *Writing custom platform-specific code.* [cit. 2022-03-07]. Available from: https://docs.flutter.dev/development/platform-integration/platform-channels?tab=type-mappings-swift-tab

[42] Google Developers. Google Developers Guide. [online], [cit. 2022-03-09]. Available from: https://developer.android.com/guide

[43] Google Developers. Bluetooth overview. [online], [cit. 2022-03-08]. Available from: https://developer.android.com/guide/topics/connectivity/bluetooth

[44] Google Developers. Wi-Fi scanning overview. [online], [cit. 2022-03-09]. Available from: https://developer.android.com/guide/topics/connectivity/wifi-scan

[45] Google Developers. Wi-Fi Aware Overview. [online], [cit. 2022-03-09]. Available from: https://developer.android.com/guide/topics/connectivity/wifi-aware

[46] Apple, Inc. App Store Review Guidelines. [online], [cit. 2022-03-08]. Available from: https://developer.apple.com/app-store/review/guidelines/

[47] Apple, Inc. About Core Bluetooth. [online], [cit. 2022-03-08]. Available from: https://developer.apple.com/library/archive/documentation/NetworkingInternetWeb/Conceptual/CoreBluetooth_concepts/AboutCoreBluetooth/Introduction.html#//apple_ref/doc/uid/TP40013257

[48] Apple, Inc. Core WLAN. [online], [cit. 2022-03-09]. Available from: https://developer.apple.com/documentation/corewlan

[49] Google LLC. Mapy Google. [online], [cit. 2022-03-09]. Available from: https://play.google.com/store/apps/details?id=com.google.android.apps.maps&hl=cs&gl=US

[50] Guilizzoni, P. What Are Wireframes? [online], [cit. 2022-04-19]. Available from: https://balsamiq.com/learn/articles/what-are-wireframes/

[51] Xia, V. A Beginner's Guide — What Is Wireframe in Software Development? [online], [cit. 2022-04-19]. Available from: https://medium.com/@Vincentxia77/a-beginners-guide-what-is-wireframe-in-software-development-60a5ab02212b

[52] Lougheed, P. Asynchronous programming: Streams. [online], [cit. 2022-04-18]. Available from: https://www.wi-fi.org/who-we-are/history

[53] Flutter Team. Interface Flutter Plugin. [online], [cit. 2022-03-09]. Available from: https://api.flutter.dev/javadoc/io/flutter/embedding/engine/plugins/FlutterPlugin.html

[54] Android Developers. WifiManager. [online], [cit. 2022-04-18]. Available from: https://developer.android.com/reference/android/net/wifi/WifiManager

[55] Android Developers. Wifi Aware Session. [online], [cit. 2022-03-21]. Available from: https://developer.android.com/reference/android/net/wifi/aware/WifiAwareSession

[56] Material Design. Flutter. [online], [cit. 2022-03-12]. Available from: https://material.io/develop/flutter

[57] Aksli, M. Material library. [online], [cit. 2022-03-12]. Available from: https://api.flutter.dev/flutter/material/material-library.html

[58] Flutter Team. sliding_up_panel 2.0.0+1. [online], [cit. 2022-03-13]. Available from: https://pub.dev/packages/sliding_up_panel

[59] Moore, K. Platform class. [online], [cit. 2022-04-27]. Available from: https://api.flutter.dev/flutter/dart-io/Platform-class.html

[60] Flutter Team. flutter_map 0.14.0. [online], [cit. 2022-03-13]. Available from: https://pub.dev/packages/flutter_map

[61] Flutter.dev. google_maps_flutter 2.1.3. [online], [cit. 2022-03-09]. Available from: https://pub.dev/packages/google_maps_flutter

[62] Flutter Campus. How to Make Google Map Autocomplete Place Search Box in Flutter App. [online], [cit. 2022-03-22]. Available from: https://www.fluttercampus.com/guide/254/google-map-autocomplete-place-search-flutter/

[63] Pub.dev. csv 5.0.1. [online], [cit. 2022-03-19]. Available from: https://pub.dev/packages/csv

[64] The Flutter Community. Read and write files. [online], [cit. 2022-03-19]. Available from: https://docs.flutter.dev/cookbook/persistence/reading-writing-files#2-create-a-reference-to-the-file-location

[65] The Flutter Community. file_picker 4.5.1. [online], [cit. 2022-03-19]. Available from: https://pub.dev/packages/file_picker

[66] The Flutter Community. share_plus 4.0.4. [online], [cit. 2022-03-19]. Available from: https://pub.dev/packages/share_plus

[67] Simform. showcaseview 1.1.5. [online], [cit. 2022-04-13]. Available from: https://pub.dev/packages/showcaseview/example

[68] Papadopoulos, M. Build and release an Android app. [online], [cit. 2022-04-15]. Available from: https://docs.flutter.dev/deployment/android

[69] Papadopoulos, M. Build and release an iOS app. [online], [cit. 2022-04-15]. Available from: https://docs.flutter.dev/deployment/ios

[70] The Flutter Community. flutter_launcher_icons 0.9.2. [online], [cit. 2022-04-15]. Available from: https://pub.dev/packages/flutter_launcher_icons

[71] Freepik. Drone Icon. [online], [cit. 2022-04-25]. Available from: https://www.flaticon.com/free-icon/drone_4212583?term=drone&page=1&position=7&page=1&position=7&related_id=4212583&origin=tag

[72] Android Developers. bundletool. [online], [cit. 2022-04-19]. Available from: https://developer.android.com/studio/command-line/bundletool

[73] Friis, S. Open Drone ID transmitter example for Linux. [online], [cit. 2022-04-23]. Available from: https://github.com/opendroneid/transmitter-linux

[74] Espressif Systems. ESP32-C3-DevKitM-1. [online], [cit. 2022-04-23]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/hw-reference/esp32c3/user-guide-devkitm-1.html

[75] Nielsen, J. 10 Usability Heuristics for User Interface Design. [online], [cit. 2022-04-27]. Available from: https://www.nngroup.com/articles/ten-usability-heuristics/

[76] Apple, Inc. How the Program Works. [online], [cit. 2022-04-30]. Available from: https://mfi.apple.com/en/how-it-works.html

Acronyms

AP Access Point. 10

API Application Programming Interface. 14, 16, 19, 21, 22, 27–29, 46, 47, 56, 57

APK Android Application Package. 50

ASD-STAN Aerospace and Defence Industries Association of Europe - Standardization. 4, 6, 10, 11, 54

BLoC Bussiness Logic Components. 17, 18

CSV Comma Separated Values. 19, 23, 24, 39, 47

DRI Direct Remote Identification. 4–7, 39

EU European Union. 1, 3–7, 58

FAA Federal Aviation Administration. 3, 4

GNSS Global Navigation Satellite System. 5

GPS Global Positioning System. 5, 11

GUI Graphical User Interface. xvii, 13, 18, 24, 42, 43, 51, 56

ID Identification. 4–7, 9, 10, 39, 51, 54, 58

IDE Integrated Development Environment. 55

IEEE Institute of Electrical and Electronics Engineers. 10

ISM Industrial, Scientific and Medical. 8

ISO International Organization for Standardization. 4

JSON JavaScript Object Notation. 39, 48

MAC Media Access Control. 10, 29, 33, 34

NaN Neighbour Aware Network. xi, 11, 18, 21, 32, 34, 39, 51

NRI Network Remote Identification. 5

OS Operating System. 18

RSSI Received Signal Strength Indication. 33, 34

SDK Software Development Kit. 15–17, 20–22, 27, 42

SSID Service Set Identifier. 10

UA Unmanned Aircraft. 3–7

UAS Unmanned Aircraft Systems. 1, 4–7

UAV Unmanned Autonomous Vehicle. 3, 5, 11, 22, 52, 54, 57, 58

UHF Ultra High Frequency. 8, 10

UI User Interface. 16–18, 20, 39, 40, 52, 53

USA United States of America. 1, 3, 4, 12

VLOS Visual line-of-sight. 4

Appendix A

Contents of enclosed CD

- `readme.txt` ..... the file with contents description and installation guide
- `build` ..... the directory with executables
- `src` ..... the directory of source codes
  - `app` ..... the directory with the application source codes
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `MP_Glejtek_Matej_2022.pdf` ..... the thesis text in PDF format