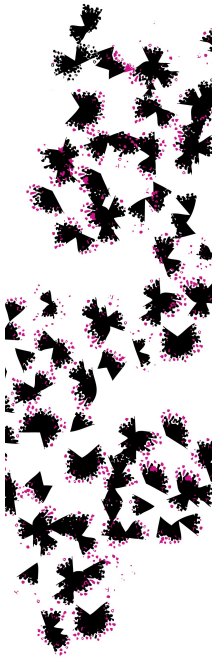

Operator control hierarchy for multiple robot systems

L. (Lan) Qin

MSc Report

Committee:
Dr.ir. J.F. Broenink
K.J. Russcher, MSc
Ir. E. Molenkamp

August 2017

041RAM2017
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA CTIT
BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

Summary

Nowadays, robotic systems are increasingly used by the Dutch National Police (NPN). The project Robots voor Veiligheid (RoVe) is aiming to address the interoperation issues of multiple robotic systems. The project uses Data Distribution Service (DDS) to enable the sharing of data among different types of robotic systems. This master assignment is focused on concerns in safely and efficiently operating multiple robotic systems.

To operate the robotic systems safely, we need to consider several situations. Conventionally, a robotic system is operated mutually exclusively by operators, which means the robotic system should only receive commands from a single operator. However, when there are multiple operators intending to operate one robotic system, unpredictable behaviours would occur because the robotic system receives all the commands from different operators. Additionally, when an operator leaves the network, due to a network or program error, or the operator exits intentionally, the robotic system will be left uncontrolled, which can also lead to unexpected behaviours. These unpredictable behaviours are undesirable and should be avoided.

On the other hand, to operate the robotic systems effectively, a stable control data model used to control the movements of the robotic systems is needed. The control data model represents the movement commands from operators, and the model needs to fit multiple types of robotic systems without modifications.

The goal of this assignment includes designing, implementing and testing an operator control hierarchy that enables safe control of the robotic systems using redundancy, designing a control data model that fits multiple types of robotic systems. The implementation should be on a Linux laptop and an Android tablet.

In this report, we elaborate the design and the implementation of the hierarchy and the control data model. In the software design, we take reusability and scalability into consideration and create a method to communicate between C++ code and Java code. This approach utilises Unix socket and protocol buffer. The method can be easily ported to languages other than Java. Finally, tests are conducted to verify the correctness and efficiency of the hierarchy. The tests are also performed to simulate the situation where multiple operators presenting in the network.

The results show that the operator control hierarchy is capable of handing over control when one operator quits the network intentionally, or the operator quits due to a program or network error. The experiments also show that the control data model can control the movements of the Eyedrive rover from the project RoVe.

Recommendations are made to improve the hierarchy and the control data model further. Mainly the further testing for both the hierarchy and the control data model. The further testing includes the measurement of detailed handover processes and the time different processes take, the measurement of response time under multiple real platforms existing in the network, etc.

Contents

1 Introduction

1.1 Context

Over recent years, development and employment of unmanned robotic systems in the military and civilian usage has emerged. This also brings concerns about these systems being fielded without sufficient capabilities to interoperate with others (Blais, 2016). The Robots voor Veiligheid (RoVe) project is aimed to address these concerns. The project is a collaboration between the Dutch National Police (NPN) and the Robotics and Mechatronics (RaM) group. It develops an architecture to enable the sharing of data between different robotic systems over Data Distribution Service (DDS).

1.2 Problem Description

Within the context mentioned above, the number of robotic systems being used by the police is increasing. This brings concerns about how to control these robotic systems safely and how to make control and command reusable.

For safely operating the robotic systems, two situations need to be considered. Conventionally, for a teleoperated robotic system like a rover or drone, the system is operated mutually exclusively, which means only one operator can control a single robotic system. The requirement of mutual exclusion brings two consequences. First, unpredictable behaviours occur when multiple operators try to control a same robotic system. Several control commands from the different operators exist in data link simultaneously. These control commands from different operators lead to unexpected behaviours. Second, when the only operator leaves the system due to a network or program error, or the operator quits intentionally, the remote robotic system will be left uncontrolled. This can also lead to unpredictable behaviours.

On the other hand, to make control and command reusable, a stable control data model used to control the movements of the robotic systems is needed. The control data model represents the movement commands from operators, and the model needs to fit multiple types of robotic systems without modifications.

1.3 Goal

To tackle the problems stated above, we first design an operator control hierarchy for multiple robot systems. The hierarchy is redundant, which means there is always one operator that controls the robotic system. When one operator quits, the next operator will seize the control of the robotic system. Furthermore, when no operator is controlling the system, the control will be automatically handed over to the onboard autonomy. Thus, the robotic system will not be left uncontrolled. This brings the fault-tolerance operation to the system.

Moreover, a control command module is designed. The control command module contains a control data model and input and output adapters. The control data model is used to control the movements of the robotic systems. The control data model is formed by a data structure representing the movement commands. No modifications need to be made for a new type of robotic system. The model makes the hierarchy compatible with the other types of robotic systems might be introduced in the RoVe project. The input adapter is on control device side, and capable of reading inputs from different input devices, e.g. joysticks and keyboards. Then, it converts the inputs into the form of the control data model. The output adapter is on the vehicle side. It adapts control data model to output signals that can be recognised by servo controllers. Then the output adapter sends the output signals to the servo controllers to control the mobility of the robotic system.

The implementation of the hierarchy and control command module will be in applications running on laptops, Android devices and vehicles' embedded computers. Tests are conducted to evaluate the correctness and efficiency of the hierarchy.

1.4 Outline

In Chapter 2, the background of DDS and Android development is given. Chapter 3 gives the analysis for common unmanned vehicles, current platforms in the project RoVe, and the requirements for the assignment. The design and implementation of the operator control hierarchy and the control command module are documented in Chapter 4. Chapter 5 shows the measurements and the results of the handover process of the hierarchy. Finally, the conclusions and recommendations are given in Chapter 6.

2 Background

2.1 Data Distribution Service

DDS is a high-performance middleware standardised by the Object Management Group (OMG). The DDS describes a Data-Centric Publish-Subscribe (DCPS) model for communications between distributed applications. The purpose of the DDS is to enable the "Efficient and Robust Delivery of the Right Information to the Right Place at the Right Time." (Object Management Group, Inc., 2017) DCPS builds on a global data space, which is accessible to all interested applications. In the global data space, following communication objects are introduced to aid the flowing of data: *Publisher* and *DataWriter* on the sending side; *Subscriber* and *DataReader* on the receiving side.

- A *Publisher* is an object for data distribution. Data published by one publisher can be various types. Applications access to a publisher via a *DataWriter*. Applications must use the DataWriter to communicate to a publisher. The DataWriter is the object to handle the value of data-objects of a given type. A *Publication* is defined as the association of a DataWriter to a publisher.
- A *Subscriber* is an object used to receive data from the publisher and present the data to the receiving application. Same as the publisher, the subscriber may receive and dispatch different types of data. A *DataReader* is an object attached to the subscriber and accesses the data from the subscriber. Thus, a *Subscription* is defined by the association of a DataReader with a subscriber.
- A *Topic* is an object that provides an identifier that uniquely identifies some data items within the global data space. All subscribers connect to the interested publishers via topic.

The conceptual overview of DCPS model is shown in Figure 2.1

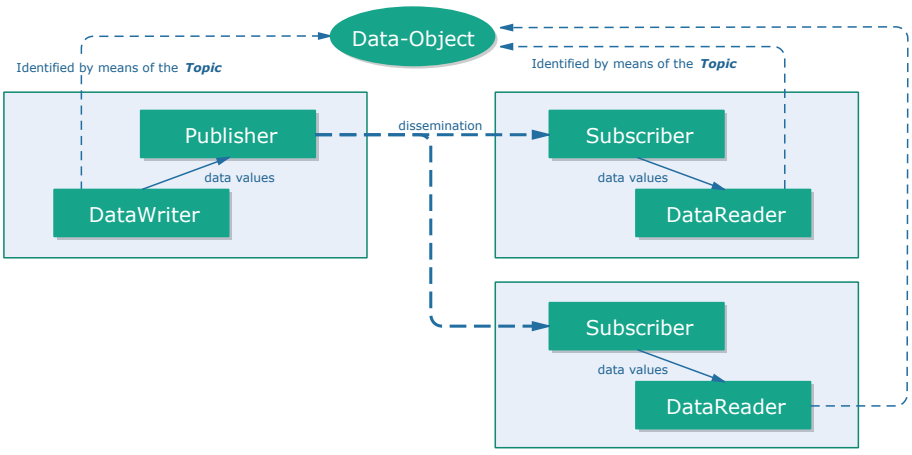

Figure 2.1: Conceptual overview of DCPS model (Object Management Group, Inc., 2017)

Moreover, supporting of Quality of Service (QoS) ensures high performance and low latency of data transformation in DDS (Object Management Group, Inc., 2017). QoS is a general concept that is used to specify the behaviour of a service. QoS contains several QoS policies. Each QoS policy is presented as a data structure formed by a name, along with a value. The fine tuning of QoS enables efficient control of a wide set of non-functional properties, such as data availability, data delivery, data timeliness and resource usage (PrismTech, 2017).

2.2 Android Development

2.2.1 Platform Architecture

Android is a software stack built for various types of mobile devices. The Android includes an operating system, middleware and key applications. The purpose of Android is to create an open source software platform for mobile network operators, original equipment manufacturers (OEMs) and developers to realise innovative ideas and building real products. These products can effectively improve the mobile experience for users.

The Android stack consists five levels as shown in Figure 2.2, the Linux kernel, Hardware Abstraction Layer (HAL), Android Runtime, Native C/C++ Libraries, Java Application Programming Interface (API) Framework, and System Applications. The three elements that are used in this assignment are the Linux kernel, the Native library, and the Java API Framework

- *The Linux kernel*. The Linux kernel is the foundation of the Android platform. The Android Runtime, which runs multiple Java virtual machines, relies on the Linux kernel for the functionalities, e.g. threading and low-level memory management. Additionally, the Linux kernel provides some of the Inter-Process Communication (IPC) mechanisms, e.g. signals, network sockets, and Unix domain sockets. We introduce the usage of the IPCs in Chapter 4.

- *Native C/C++ Libraries*. Many core Android system components and services are built from native code that requires native libraries written in C and C++. In this assignment, we build OpenDDS library based on these native libraries. In the following chapter, this layer is referred to as the native layer.

- *Java API Framework*. The most featured Android APIs are written in the Java language. In the following chapter, we refer this layer to as the Java layer. These APIs form the building blocks which can be used to create Android applications. The usage of these APIs simplifies the reuse of core, modular system components and services. These components and services include:

  - *View System* to build an application's Graphical User Interface (GUI),
  - *Resource Manager* to provide access to localised strings, graphics, and layout files,
  - *Activity Manager* to manage the life cycle of applications and provide a navigation stack for switching between different screens.
  - *Notification Manager* and *Content Providers* are unused in this assignment. Therefore, they are not introduced here.

2.2.2 Development Kits

For the development based on the native layer and Java layer, Google offers different development kits, Native Development Kit (NDK) and Android Software Development Kit (SDK). NDK enables to use native C/C++ code in applications, while SDK enables developers to write code in Java.

Despite Java is mostly used in Android development since the code related to user interface is required to be written in Java, and most functionalities provided by Android are presented in Java APIs, native C/C++ code is necessary under situations where:

- the code may be compute-intensive and have requirements of efficiency,

- the C/C++ code may already exist,

- the code may need to be shared with other platforms, etc.

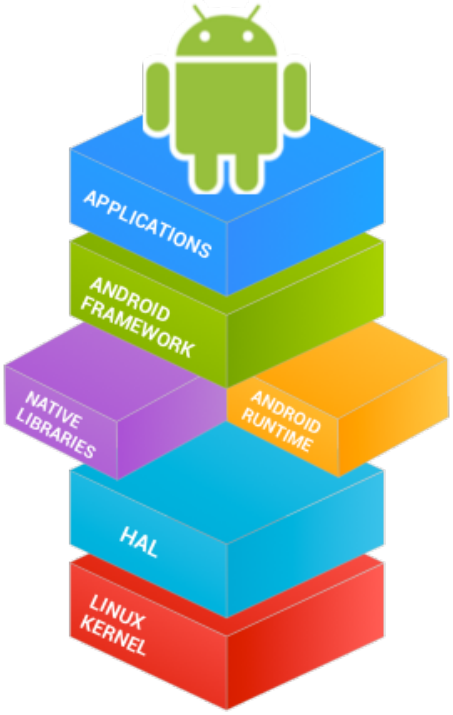

Figure 2.2: Android stack (Google Inc., 2017a)

In this assignment, the DDS implementation we used is written in C++, and considering portability to other platforms, we utilize both SDK and NDK in the design.

2.3 Current Platforms in RoVe

Currently, different platforms are presented in the project RoVe. The differences are in types of vehicles, onboard embedded computers, and control devices. Figure 2.3 shows an overview of the platforms. The arrows indicate data exchange. In this assignment, the embedded computer in the vehicle only sends set points to the servo controller. Therefore, the arrow between embedded computer and the servo controller is unidirectional, while all the other arrows are bidirectional. Additionally, Table 2.1 shows the alternatives in different platforms.

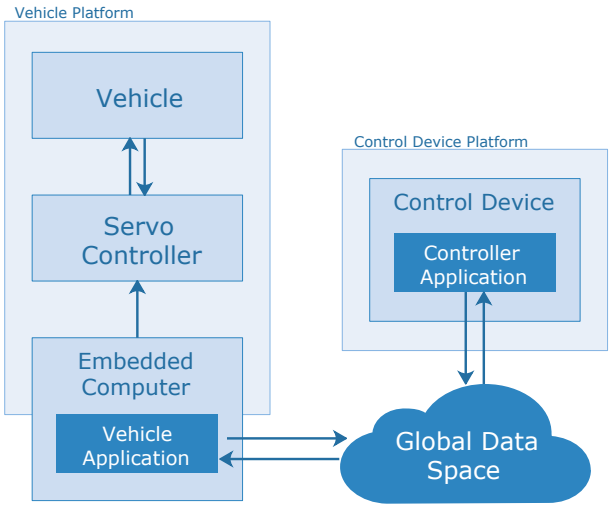

Figure 2.3: Overview of platforms

Table 2.1: Alternatives in platforms

	Alternatives	
Vehicle	DJI Phantom 3	Eyedrive
Servo controller	DJI NAZA-M	Eyedrive servo controller
Onboard embedded computer	Raspberry Pi, RaMstix	
Control device	Linux laptop	

3 Analysis

3.1 Common Unmanned Vehicles

Nowadays, unmanned vehicles play increasingly important roles in modern society. In the field of law enforcement, they are also increasingly used for surveillance, search, and rescue, even riot control and chemical, biological agent detection. The deployment of the unmanned robotic systems in these tasks results in lower cost per flight hour than a manned helicopter, and lower risk of human life in dangerous operations.

In general, the unmanned vehicles have three different modalities based on where they operate (Murphy, 2014). If they operate

- *on the ground*, they are called Unmanned Ground Vehicles (UGVs);
- *on the water*, they fall into the general category of Unmanned Surface Vehicles (USVs);
- *in the air*, they may be called by various names including Unmanned Aerial Vehicles (UAVs), Unmanned Aerial Systems (UASs).

We focus on these remotely operated vehicles. The remotely operated vehicles are the vehicles which are controlled by an operator who is not in the vehicle. These vehicles can be operated by radio control, or through a cable or line connecting the vehicle to the operator's location. Besides, we only focus on the situation where the operators direct the vehicles via hand-controllers, e.g. 3-axis joysticks for vehicle rates. Autonomy is out of the scope of this assignment.

In this assignment, there are several roles in a remotely operated system. We define them here, and they will be commonly used in the rest part of this report.

- *Operator* is a person who monitors the operated machine and makes the needed control actions.
- *Controller* is an electronic hardware device that is used for receiving data from operators and sending to vehicles, also displaying information from the network. For example, a controller can be a laptop, a tablet, or a mobile phone. We will also call it control devices in following texts. Notice that the controller is not the servo controller in control engineering, and in this report, the controller is only referred to as the control device.
- *Servo controller* is an loop controller that is responsible for the hard-time control of servos in the vehicle.
- *Input device* is a device that generates movement commands from the operators' inputs. For example, an input device can be a joystick or a keyboard.
- *Vehicle* is a mobile robotic system that receives the commands from the terminals and executes these commands to perform corresponding motions.

3.2 Current Work in Project RoVe

As a part of project RoVe, this assignment shares some preconditions and requirements from RoVe. To clarify them, we analyse the context of the project in this section.

3.2.1 Data Distribution Service

There are growing types and numbers of unmanned vehicles being used by the police, as previously mentioned. This brings concerns in the data communication. The difficulties can be broken down into following listed aspects (Pardo-Castellote, 2010).

- In vehicle platforms, the difficulty is that these platforms are mostly embedded and hence the power and memory resources are constrained.
- In data-link communications, the difficulty is that there might be many types of traffic, e.g. sensor data, command and control data, and status, mission, supervisory data. Also, the urgency, priority, reliability and volume for these different types of data vary from each other. Moreover, there are challenges in the communication channels, such as the channels with significant latency or low throughput.
- In controllers, the difficulties mainly fall in heterogeneous systems, multiple programming languages and requirements of reconfiguration.

These difficulties result in that it is time-consuming, error-prone, and inefficient to create an own data-link protocol. The introduction of middleware is aiming to tackle these problems. Middleware is computer software that provides services to software applications beyond those available from the operating system. It can be described as "software glue". There are two main reasons for using middleware (Mohamed et al., 2008):

- achieving reusability of logic through a standardised, modular programming model,
- application developers being able to get rid of the underlying data distribution.

Currently, the Robot Operation System (ROS) is one of the most mainstream open-source robot middlewares in the field of robots. Compared to ROS, DDS has following advantages.

- The introduction of distributed network architecture in DDS eliminates the needs of central servers. A crash of single application will not influence the communication of other applications in the network. Thus, a single-point failure will not happen in DDS.
- DDS uses Quality of Service to configure the performance of the communication. This makes the communication more flexible.
- DDS is based on UDP/IP protocol. Multicast transmission greatly increases the network throughput and the real-time performance of the communication.

In the project RoVe, considering the advantages listed above, along with security, manageability, and interoperability issues, DDS was chosen to be the middleware in the whole architecture. Therefore, designing and implementing software based on the DDS is one of the requirements of this assignment.

3.2.2 Tasks in This Assignment

The increasing number of robotic systems brings two problems: how to control vehicles safely, and how to efficiently control the movements of vehicles.

For the first problem, redundancy is one of the approaches that increase the reliability of vehicle control. In engineering, redundancy is the duplication of critical components or functions of a system with the intention of increasing reliability of the system. In RoVe project, the feature of multiple operators makes it possible to introduce redundancy in the network.

There should always be control devices controlling the vehicles to prevent vehicles from being uncontrolled. Therefore, under this situation, the critical components are the control devices.

Redundancy is formed by control devices that multiple human operators held. When one control device fails to send commands due to unstable network, program errors, or the battery running down, one of the other control devices will step in and take over the control of the vehicle. The embedded computer in vehicles can also be a control device to control the vehicles when there is no other control device in the network, or when the vehicles drop out of the network. These behaviours ensure that the vehicles are always under control. Designing and implementing this redundant operator hierarchy is one of the issues need to be solved in this assignment.

On the other hand, the fixed operator hierarchy is sometimes not enough for performing a task. A handover scheme is needed in the hierarchy. Consider a situation where multiple operators are performing a task on a single robotic system, and the task requires collaborations between different operators. The control might need to be handed to another operator sometimes. However, the handover is only performed when control devices join or quit the network. Therefore, to hand over control to a specific device without quitting the network, a handover scheme is needed.

For the second problem, the types of the vehicles are also increasing in the RoVe project. To make the data distribution scalable in the future, we need to design a stable control data model. The control data model represents movement commands from the operators. The design should take the universality and stability into accounts, which means the control data model should represent the movement commands for different types of vehicle with an identical interface.

3.3 Requirements

Based on the analysis and discussion above, we derive requirements for this assignment. The requirements are divided into several parts: the operator control hierarchy, the control command module, the implementation and tests, and the general requirements.

3.3.1 Operator Control Hierarchy

Requirement 1: *The hierarchy must exploit DDS standard as much as possible.*

The design should rely on DDS standards. The DDS standards provide abundant QoS that can be used. Using these standards and QoS can minimise the occurrence of errors in the program and decrease the development time.

Requirement 2: *The hierarchy must include a handover scheme.*

As it mentioned in the last section, it is not possible to rely only on DDS standards in some circumstances. The preemption of control for operators is necessary. A handover scheme must be designed to handle the preemption.

3.3.2 Control and Command Module

Requirement 3: *The control and command module should contain a control data model that can be used to control different types of vehicles with identical data structure.*

The control data model provides a universal control data structure for representing the operators' movement commands. In project RoVe, new vehicles might be introduced in the future, e.g. boats, fixed-wing UAVs. The control data model needs to control movements of these vehicles without modification of the data structure.

Requirement 4: *The control and command module could contain an input adapter that takes inputs from different devices and converts the inputs into the form of control data model.*

Requirement 5: *The control and command module should contain an output adapter that converts the movement commands in the form of control data model to the outputs that can be recognised by the servo controller.*

Different user input devices, e.g. joysticks, keyboards, or virtual joysticks could be taken into consideration. The input adapter is capable of reading from these input devices and converting these inputs into the form of the control data model, while the output adapter is capable of converting the control data model to the servo control signals.

3.3.3 Implementation

Requirement 6: *The implementation must be built based on OpenDDS.*

The OpenDDS is an open source DDS implementation. Compared to other commercial implementation of DDS standards, the open-source feature of OpenDDS ensures that we can compile the OpenDDS library freely and deploy our applications on different platforms without customization from providers. Thus, it brings more flexibility to the development. In project RoVe, OpenDDS has been chosen as the DDS implementation, therefore, using OpenDDS is a requirement of this assignment.

Requirement 7: *The implementation must run on a Linux machine.*

Requirement 8: *The implementation should run on an Android device.*

Currently, there is no document on how to develop the OpenDDS applications on Android devices. This adds uncertainties to the assignment. However, the implementation will be running on a Linux machine at least.

Requirement 9: *The implementation must have a graphical user interface for demonstrating the use of hierarchy and testing.*

A graphical user interface must be presented to demonstrate the main functions of the hierarchy.

3.3.4 Tests

Requirement 10: *Tests must be performed to evaluate the correctness of the hierarchy.*

The correctness of the hierarchy is the first thing needs to be tested. These tests are aimed to ensure the handover process will eventually be done in good network conditions.

Requirement 11: *Tests should be performed to evaluate the robustness of the hierarchy.*

Fault-tolerance is one of the responsibilities of the hierarchy, so robustness is critical and needs to be checked. For example, the handover process should occur when a program crashes, or network fails.

Requirement 12: *Tests could be performed to evaluate the correctness of the control data model.*

This includes the test for different input devices, also the test for different types of vehicles.

Requirement 13: *Tests should be able to perform without being connected to real vehicle platforms.*

It is not necessary to use the actual vehicle platforms to do the test. A laptop can be used to simulate vehicles and check the correctness of the commands.

Requirement 14: *Tests won't be carried out on platforms that currently not presented in the RoVe project.*

The interfaces for other platforms will be taken into account in the design, but they will not be implemented or tested.

3.3.5 General Requirements

Requirement 15: *Reusability, scalability, and extensibility must be considered in hierarchy and software design.*

Object-oriented and certain design patterns should be introduced during design and coding to make the software open for extension.

Requirement 16: *The software must be light-weight.*

Follow the "Don't repeat yourself" rule and properly utilise third-party software.

4 Design and Implementation

4.1 Operator Control Hierarchy

The design of the operator control hierarchy is broken down into two parts, static structure, and dynamic behaviour. In the static structure design, we first categorise types of operators and control devices and put them on different levels. Then, in each level, operators are further categorised by the controller devices they use. For the dynamic behaviour, we design the hierarchy based on the DDS standard and give explanations about the handover procedure in different situations.

4.1.1 Static Structure

Operators are categorised by to what extent are the operators interested in controlling and getting information from the target unmanned vehicles. The operators are divided into three types.

The first operator type is the mobile operator. The mobile operators have the most interest in controlling the vehicles. These operators can be individual operators or operators in mobile operating units, e.g. operating vans. They manipulate the vehicles per instant visual feedbacks. They are the operators with most sufficient control information and feedbacks. These operators can operate the vehicles more safely. Therefore, they have the highest priority when other types of operators are presenting in the network.

The second operator type is the stationary operator. The stationary operators are the operators in local control stations. They are not required to be in the visual range to the target vehicle. If they are in the visual range, the vehicles are controlled directly by visual feedbacks, as the mobile operators. When locations of local control stations are without the visual range, the operators will control the vehicles via feedbacks from onboard cameras or other sensors. This makes stationary operators less informative than the mobile operator. These operators are less interested in control of vehicles. Thus, the priority is less than mobile operators.

The third operator type is the remote operator. The remote operators are the operators without visual range. They can be in a control centre in e.g. another city. They are less interested in directly controlling the vehicle than the stationary operator. Therefore, the priority of the remote operator is the lowest among human operators.

The vehicle onboard controller is considered as the safety controller in the bottom of the hierarchy. The onboard controller seizes the control of the vehicles when no other active control device. This controller ensures that vehicles are always under control. When there are no other control devices in the network, this controller simply hovers the vehicles or runs an autonomy algorithm that directs the movement of the vehicles. This controller has the least priority and acts as the last protection of the vehicles.

Operators with same priority but different control device also need to be considered. Control devices also need to be categorised. We design that control devices with higher portability have higher priority since operators with more portable control devices can move easily to get better feedbacks from the vehicles. We list some types of the control devices here, such as mobile phones, tablets, laptops, and desktops.

Table 4.1 shows the design of the static structure.

4.1.2 Dynamic Behaviour

First, the hierarchy has close connections to the `OWNERSHIP` QoS in DDS. The exclusive mode of `OWNERSHIP` along with `OWNERSHIP_STRENGTH` ensures only data writers with highest

Table 4.1: The Static Hierarchy

Operator types	Controller devices	
Mobile operators	Handheld	Phones
		Tablets
		Laptops
	In a operating unit	Laptops
		Desktops
Stationary operators	Phones	
	Tablets	
	Laptops	
	Desktops	
Remote operators	Phones	
	Tablets	
	Laptops	
	Desktops	
Vehicle onboard controllers	Onboard controllers	

`OWNERSHIP_STRENGTH` could write to the topic. In other words, these features make sure that only the control device with the highest priority can write control commands to the vehicle. The mutually exclusive control of one vehicle relies on these features.

Before we discuss the hierarchy further, there are two concepts, priority and handover sequence. The priority is related to preemption. The control devices with higher priority can seize the control from other control devices with lower priority. While the handover sequence is related to the action when one controller quits the hierarchy. Therefore, we use priority for handling the controller join and handover sequence for controller quitting.

We can divide the hierarchy into two dimensions, as it is shown in Figure 4.1. We assume all the devices on each layer are intended to control Vehicle #1.

Vertically, there are different layers. In the bottom of the hierarchy, it is the onboard controller. As it mentioned in the last subsection, this layer is designed to handle the situation where no controller in the network is controlling the vehicle. Different from other layers, this layer is in the vehicle rather than the control devices, and we consider this layer is the last protection to prevent the vehicle from being uncontrolled. The other layers in the hierarchy are built in control devices. The higher layers have more priority than the lower layers. When control device with higher priority joins, it seizes control from the lower layer devices automatically.

Horizontally, in each layer, there is only one type of control devices. For example, in Layer #2, they are all tablets. On the same layer, all the devices have the same priority. When one of these control devices quitting the network, the control will be handed to another control device in the same level automatically.

Consider one situation, in the network, Operator #1, #2A, and #2B are using different control devices. All of them are trying to control Vehicle #1.

In this situation, Phone #1 in the Layer #3 has the highest priority in the hierarchy. Thus, Phone #1 seizes the control of the vehicle from Tablet #1.

When Phone #1 quits the network, the control will be handed over by the hierarchy. According to the handover sequence, the control will be handed to Layer #2. Tablet #1 or #2 will seize the control. When Tablet #1 quits, the control will be handed to Tablet #2.

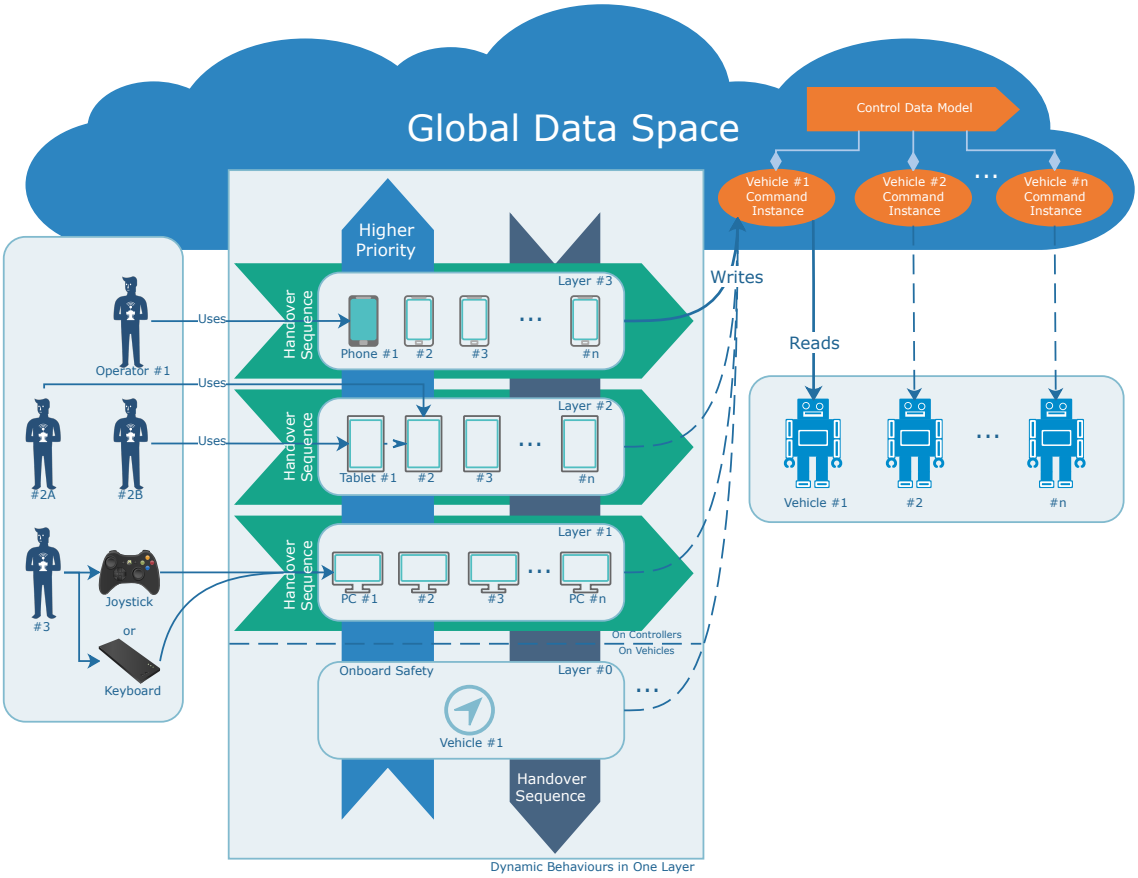

Figure 4.1: Dynamic behaviours in each layer

When no device tries to write the command topic in the network, the onboard controller seizes the control automatically.

4.2 Control and Command Module

4.2.1 General Design

As it stated in subsection 3.3.2, the control and command module consists of the control data model and the adapters. An overview of control and command module is shown in Figure 4.2.

The control data model is defined as a data structure used to control the vehicle movements. Vehicles application reads and decodes control data model and converts to signals to be sent to the servo controller. Different types of vehicles share an identical control data model.

For the user input messages, various devices are used to generate such messages, such as joysticks or keyboards. To handle these different devices, we need certain interfaces and adapters to isolate changes.

The adapters consist of input adapter and output adapter. For the input adapter, it only exists on control devices. The input adapter reads inputs from various input devices, then converts the inputs into the form of control data model. The output adapter only exists in the vehicles. It is the bridge between the control data model and the inputs of the servo controller. The output adapter converts the movement commands in the form of the control data model to the signals that the servo controller can be recognised.

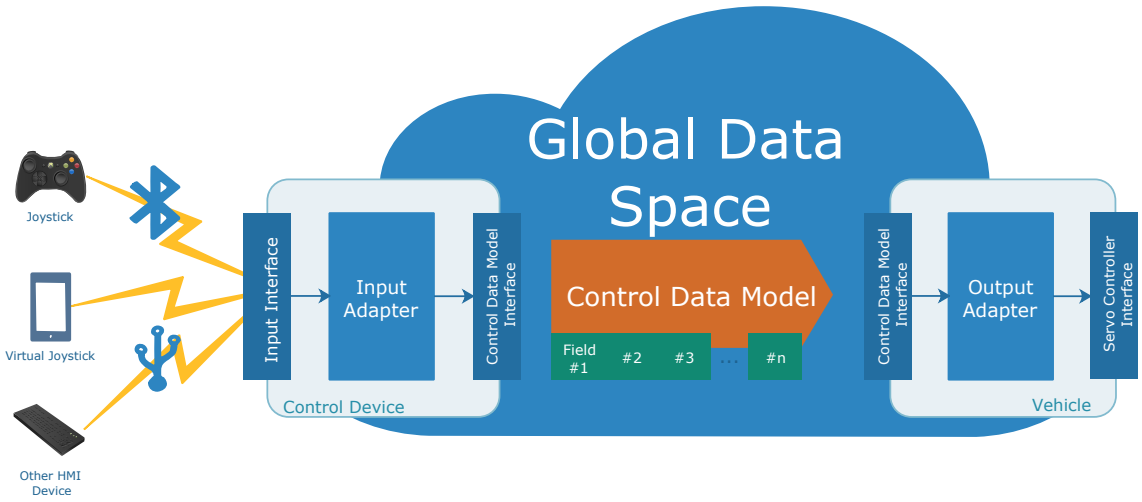

Figure 4.2: Overview of control and command module

4.2.2 Control Data Model Design

The common vehicles have been listed in section 3.1 and they are UAVs, UGVs and USVs. To design a control data model, we need to take these vehicles into considerations at least. As the control data model is used to control the movements of the vehicles, we focus on the movement command design in this subsection.

UAVs move in three-dimensional space, while UGVs and USVs move in two-dimensional space. We cannot control the position of UGVs and USVs in the z-axis. To design a control data model that fits both UAVs, UGVs and USVs, we can first design for UAVs and then modify it to make it suitable for UGVs and USVs, since UGVs and USVs are moving in a two-dimensional space, and we can easily reuse UAVs movement commands and ignore the control commands on the z-axis.

The movement commands are sent to the servo controllers in the vehicles. Since control data model is used on the top of the servo controllers, we use the interfaces of the servo controllers as the start point of control data model design.

4.2.3 Movement commands for UAVs

Currently, multi-rotors are mostly used in the drone world. Two UAVs that used in the project RoVe now are multi-rotors. However, there are also other types of UAVs. The typical types or designs of UAVs include fixed-wing, multi-rotor VTOL (Vertical Take-Off and Landing), nano, hybrids and others. In 2015, multi-rotor drone types accounted for a market share of more than 75 percent in the global commercial drone market. Besides, fixed-wing drones are the other major drone type (GPS World, 2016). Based on these data, we assume the mainstream types of UAVs are multi-rotor and fixed-wing. So, in the design, we consider these two kinds of UAVs.

To gain general control interface of various UAVs, we analyse DJI Naza-M multirotor autopilot system currently used in the project RoVe, along with popular open-source unmanned vehicle autopilots from both ArduPilot (ArduPilot Dev Team, 2016a) and PX4 (Meier et al., 2015). The Naza-M is only for multi-rotor UAVs, while ArduPilot and PX4 are suitable for both multi-rotor and fixed-wing UAVs. Notice that the ArduPilot and PX4 also support for the rover control. We can derive the control interface for controlling UGVs or USVs from these autopilots.

DJI Naza-M multirotor autopilot Naza-M is the autopilot that currently used on the DJI Phantom 3 drone in the project RoVe. The movement control inputs of Naza-M are listed below.

- Roll
- Pitch
- Rudder (yaw)
- Throttle
- Additional control for mode switching, gimbal control, and others are the platform-specific functions

ArduPilot ArduPilot is used for multi-rotor and fixed-wing drones, and it also supports ground vehicles and boats. ArduPilot receives movement commands from radio control (RC) receiver. We derive the control interface of vehicles by listing the RC inputs. The RC inputs used in ArduPilot (ArduPilot Dev Team, 2016b) are listed below.

- Roll
- Pitch
- Throttle
- Yaw
- Flight modes
- (Optional) Inflight tuning or camera mount

Similar as Naza-M, the movement commands are throttle, pitch, and yaw. Besides movement commands, there are also mode switch, camera mount control, and platform-specific commands.

PX4 The RC inputs used in PX4 (Meier et al., 2017) are listed below.

- Throttle
- Yaw (Rudder)
- Roll (Ailerons)
- Pitch (Elevator)
- Mode switch (Manual/Auto)
- Function switch (Activation/Deactivation of individual stabilization functions)

Although at least six channels are needed, the movement control only relies on throttle, yaw, roll, and pitch. The rest channels are similar to Naza-M and ArduPilot. There have no camera control commands on PX4 autopilot.

From the three different autopilots presented above, we can conclude that the movement commands are the same, and they are the throttle, pitch, roll, and yaw. Notice that the ArduPilot and PX4 support both multi-rotor and fixed-wing UAVs, so we assume that both multi-rotor and fixed-wing UAVs share a similar control interface.

Therefore, we use quadruple formed by the throttle, pitch, roll, and yaw as movement command interface for UAVs.

Besides, there are also additional commands in various autopilots. These additional commands can be divided into two types, the mode switch commands and the camera manipulation commands. We also take these two types of additional commands into considerations in the following design.

4.2.4 Movement Commands for UGVs and USVs

As it stated above, when the control command interface for controlling UAVs is chosen, the movement command of UGVs and USVs can be easily derived, since UGVs and USVs move in two-dimensional space. In this part, we will consider UGVs first, since UGVs have already presented in RoVe, and UGVs and USVs are very similar when it comes to control. In this subsection, we analyse the current platforms in the RoVe project. Then, both the ArduPilot and PX4 autopilots support for rovers, so that we review these two autopilots as well.

Eyedrive Steering, throttle

ArduPilot Steering (roll), throttle, and flight modes

PX4 Steering (roll), throttle

From the examples listed above, for UGVs and USVs, the control signals are throttle and steering values. However, there are differences in how to map steering values into the aforementioned quadruples formed by the throttle, pitch, roll, and yaw.

Intuitively, the steering values should be mapped into yaw, since the turning of vehicles is rotating around the z-axis. The reason ArduPilot and PX map steering to roll channel might be that, in standard RC transmitter, the throttle and yaw are on the same stick. If the steering maps to yaw, the throttle and steering will be manipulated by only one hand. This is hard to operate. Therefore, the steering is assigned to roll, as we can control the throttle and steering separately by two hands.

Compared to the pitch, the roll is more intuitive since the steering will direct the vehicles into left or right, in standard RC transmitter, the roll is changing in horizontal direction, while the pitch is in the vertical direction. Therefore, the steering is mapped to roll rather than pitch.

Under considerations above, we use quadruples formed by the throttle, pitch, roll to control the UGVs and USVs. However, the pitch and yaw are ignored in the UGVs and USVs control.

4.2.5 Additional Commands

As it mentioned in section 4.2.3, except the movement commands, there are also additional commands we need to consider. These additional commands can be categorised into camera manipulation commands and mode switch commands.

Despite the RoVe project mainly focuses on the surveillance and observation in law enforcement agencies, camera control commands are not be included in the control data model, because the model is only responsible for the vehicle mobility. It is more reasonable to create another data model for the payload manipulation.

The other type of additional commands is the mode switch commands. Different autopilot providers define these commands. Therefore, these mode switch commands vary from each autopilot from different providers.

It is difficult to conclude all commands that different providers give, so we do not provide an enumeration that includes all of these commands. We conclude the modes that used in NazaM, ArduPilot and PX4, and they are the manual mode, assisted mode and automatic mode.

For different autopilots, there are sorted modes. However, these finely sorted modes should be adopted to specific models of autopilots when introducing new vehicles. It is developers' responsibilities to perform the formatting and parsing of the enumeration for the specific platforms.

4.2.6 Control Data Model

According to the discussion above, a control data model is designed as below.

```
enum Command_type_t {
    PROBE,
    MOVEMENT,
    MODE
};

enum Mode_t {
        MANUAL = 0,
        ASSISTED = 10,
        AUTO = 20
}

struct MovementCommand {
    short throttle;
    short roll;
    short pitch;
    short yaw;
};

struct ControlCommand {
    long long src_id;
    long long dest_id;
    Command_type_t command_type;
    MovementCommand movement_command;
    Mode_t mode;
    TimeBase::TimeT timestamp;
};
```

Notice that the throttle, roll, pitch, and yaw are signed short type. Therefore, the value of these numbers ranges from -32767 to +32768. In this control data model, there is also some information that related to other topics, such as source ID, destination ID, and timestamp. These parts will be explained in following topic design section.

4.3 Topics

In the DDS, topics are the fundamental means of interaction between the applications in the network. The goal of the topic design is to meet the requirements of the system. Therefore, we use the requirements along with the functions that we need to realise as the entry point of the topic design.

According to the analysis and requirements in Chapter 3, the main functions can be categorised into following types:

- displaying devices that currently in the network,
- sending or handling requests for controlling specific vehicle,

- sending or handling control commands.

According to these functions, we designed three topics to handle these businesses. These three topics are listed below.

- `DeviceInfo` topic. Used to maintain a list of the current online devices and inform the changes in the ownership relations.
- `RequestControl` topic. Used to pass the requesting and replying messages for the control rights.
- `ControlCommand` topic. Used to deliver the commands to the vehicles.

In the following subsections, the detailed design of each topic will be given.

4.3.1 Important Concepts

Before we elaborate the design of the topics, several important concepts need to be clarified. These concepts include keyed topic, content-filter topic and some QoS policies used in the topic design.

Keyed topic Each topic data type can specify zero or more fields that make up its *key*. In DCPS terminology one can publish individual data samples for different instances on a topic (Object Computing, Inc., 2017b). Each instance is associated with a unique value for the key. If a topic is keyed, the instances can be independently maintained, and the key forms the unique identifiers of data objects.

Content-filtered topic A content-filtered topic is a topic with filtering properties. The usage of content-filtered topic enables us to only get the data that we are interested in from the topic.

For example, consider we have a topic that contains control commands, but vehicles are only interested in the control commands whose destination is the vehicles. The content-filtered topic can filter out all the samples that are not sent to the target vehicle.

Important QoS policies Following QoS policies are used in this assignment.

- *Durability*. The `Durability` policy specifies whether data writers should keep samples after the samples have been sent to known subscribers. That is, whether the late-joining data readers can receive the previous samples published in the data writer.
- *Liveliness*. The `Liveliness` policy controls when and how DDS services determine the availability of participants changes. If we set the `Liveliness` to `AUTOMATIC`, the DDS service will send a liveliness notification if the participant has not sent any network traffic in last `lease_duration`.
- *Ownership* and *Ownership_strength*. The `Ownership` policy specifies whether multiple data writers can write samples of the same instance. If `Ownership` is set to `EXCLUSIVE`, only one data writer is allowed to write to the same instance. The data writer is elected by value of the `Ownership_strength` policy. The data writer with highest `Ownership_strength` owns the instance.

4.3.2 DeviceInfo Topic

In a network with multiple controllers and multiple vehicles, for an operator, it is necessary to be provided with intuitive information about which vehicles are under control. Otherwise, it can be dangerous to send a command to a random vehicle. Also, to control a specified vehicle that currently controlled by another controller, the requester should send a request to

the vehicle asking for control. This operation cannot be done without knowing the identification of the vehicle. Therefore, a unique identification for both controllers and vehicles in the network is needed.

It is also required to know which controller is currently controlling a specified vehicle. An operator with high priority can control several vehicles at the same time. The `Ownership` QoS policy of DDS guarantees this behaviour. However, DDS does not explicitly provide APIs to check which data writer is the owner of a sample in topics. This is an important reason to create a `DeviceInfo` topic to store the information of ownership relations, i.e. which controller currently controls this vehicle. The detailed way of how to realise this function will be given in following section 4.3.6 since this function needs collaborations with `ControlCommand` topic.

To conclude, the reasons for building this `DeviceInfo` topic is, first, one device needs a unique identification in the network. Second, the operators need to recognise the ownership relations between the vehicles and controllers explicitly.

To satisfy the functions mentioned above, we design the `DeviceInfo` topic as follow.

```
enum Device_type_t {
    VEHICLE,
    CONTROLLER
};

struct DeviceInfo {
    long long device_id;
    string name;
    Device_type_t device_type;
    long long controller_id;
};
```

Notice that the `controller_id` field is only applicable to vehicles, and it indicates which controller is currently controlling the vehicle. For controllers, this field is left unused in the program.

Each device should maintain one sample in this topic. Therefore, the key of this topic is the device ID. Besides, the QoS settings need to be configured in this topic.

`Durability` QoS policy The default kind of this QoS is `VOLATILE_DURABILITY_QOS`. Under this default QoS, DDS does not store the samples after they are delivered. Therefore, late-joining data readers will not get any information from the data writer that already online. However, the late-joining data readers should also get the information from all the currently online devices to generate an online device list. Thus, the `Durability` QoS needs to be set to `TRANSIENT_LOCAL_DURABILITY_QOS`. Under this QoS, the DDS will attempt to keep samples so that they can be delivered to any potential late-joining data readers.

`Liveliness` QoS policy The default kind of this QoS is `DDS_AUTOMATIC_LIVELINESS_QOS`, which means DDS will automatically assert liveliness for the DataWriter at least as often as the `lease_duration`. However, the default `lease_duration` of this QoS is infinite, which means DDS will not check the liveness of the data writers. This causes problems when a DDS program quits abnormally, or network blocked. The DDS does not assert the liveliness of these data writers and the DDS will assume the data writers in this program still alive. The `lease_duration` of `Liveliness` QoS policy need to be set to a certain value rather than infinite, say 1 second. Under this setting, the DDS will detect the liveliness of data writer per second. Other data readers will detect program crashing or network interruption. As for the

exact value of `lease_duration`, it depends on the users' specific requirements. We set `lease_duration` to one second here.

4.3.3 RequestControl Topic

This topic is to pass the messages for requesting for control. In a requesting and answering handover process, there are three roles as it listed below.

- *Requester*. The controller asks for the control of a vehicle.
- *Replier*. The controller currently controls the vehicle.
- *Vehicle*.

The designed process of the active handover is shown as followed.

1. The requester sends a request. The request includes the ID of the vehicle that the requester intends to control, the ID of the replier, and the ID of the requester. The replier is the current controller of the vehicle. The ID of the replier can be derived by checking the local device list.
2. The replier receives the request, waiting for the operator accepting or declining the request. If the operator declines, the handover will end.
3. If the operator accepts, the replier will publish an accept message in the network to inform the requester to take actions.
4. The requester receives the response and takes steps to seize the control of the vehicle.

The detailed actions of the replier and requester take after operator accepting are given in the following subsection 4.3.4, since it is related to the `ControlCommand` topic.

According to the discussions above, we give the static structure of this topic here.

```
enum ack_t {
    REQUEST,
        ACCEPT,
        REFUSE,
        FINISH
};

struct RequestControl {
    long long src_id;
    long long dest_id;
    long long current_controller_id;
    long src_priority;
    ack_t ack;
    TimeBase::TimeT timestamp;
};
```

When a requester wants to send a request, the only information needs to be provided by the user is the `dest_id`, i.e. the ID of the vehicle that requester intends to control. Since the program stores the current device information of all the devices online, the `current_controller_id` of the vehicle can be derived from the device list. Finally, the program will automatically add the device ID of itself and then publish this request in the network.

When a replier receives and accepts the request, the replier will modify the `ack` field of the request message to `ACCEPT` and publish this response to the network. The requester will receive this message and take actions.

However, not all devices need to receive this message. Only devices with ID equal to the `current_controller_id`, or `src_id` (to receive responses) need to receive the messages from this topic. Therefore, we need to set this topic to a content filtered topic to filter irrelevant messages.

The `src_priority` is a reserved field currently. This field is supposed to realise a priority checking function in the future, i.e. if the `src_priority` is lower than current controller's priority, the current controller will directly hand over the control to the requester without asking the permission from the operators.

The QoS of this topic is set to default.

4.3.4 ControlCommand Topic

This topic is to deliver commands to vehicles. Moreover, it is the core part of handover. The interface description of this topic is intuitive. The description is shown in previous subsection 4.2.6. Except for the movement commands that control the vehicles, there are only `src_id` and `dest_id`. The `src_id` is the controller ID, and the `dest_id` is the vehicle ID. The control commands are sent from `src_id` to `dest_id`.

When multiple controllers write to one vehicle, the vehicle should not take all the messages, but only the messages from the owner of this sample. The owner is the data writer with the highest priority. The DDS's `Ownership` QoS policy ensures this mutually exclusive behaviour. Moreover, `Ownership` applies on a per key value basis for keyed topics. This means if a topic is keyed, every instance in the topic has an owner. Since one vehicle should only take commands from one controller, the key of the topic should be set to the vehicle's ID, i.e. the `dest_id` field of the interface description. So that for each vehicle, multiple controllers can write to it, but the vehicle only takes commands from the controller with the highest priority.

In this way, the DDS ensures the passive handover. However, the situation is different under the active handover. Consider a situation where there are two controllers and two vehicles in the network, as shown in Figure 4.3.

In this figure, assume the data writer in Controller 1 has higher `OWNERSHIP_STRENGTH` than data writer in Controller 2. After all the devices become online, Controller 1 will seize the control of Vehicle 1 and Vehicle 2, since Controller 1 has the highest `OWNERSHIP_STRENGTH` in the network.

The solid lines in the figure indicate that this data writer is the owner of the data reader in the vehicle. The dotted lines indicate that the data writer is not the owner of the data reader. For these dotted lines, although the writer can write to the reader, the reader will drop the sample sent by this data writer. When the other controllers quit the network, these controllers with dotted lines have the chances to be the new owner of the vehicle. They are the redundant controllers in the network.

Consider the situation where Controller 2 now wants to control Vehicle 2. Some steps should be taken here to increase the `OWNERSHIP_STRENGTH` of the data writer in Controller 2. If we directly increase the `OWNERSHIP_STRENGTH` of the data writer in Controller 2 and make it larger than the data writer in Controller 1, the data writer in Controller 2 will also seize the control of the Vehicle 1 since it has higher `OWNERSHIP_STRENGTH` now. This is not ideal since we only intended to seize the control of Vehicle 2.

To solve this problem, we separate the data writers into two kinds, fail-safe writer and priority writer. The fail-safe writer is the default writer of the controller. One fail-safe writer can write

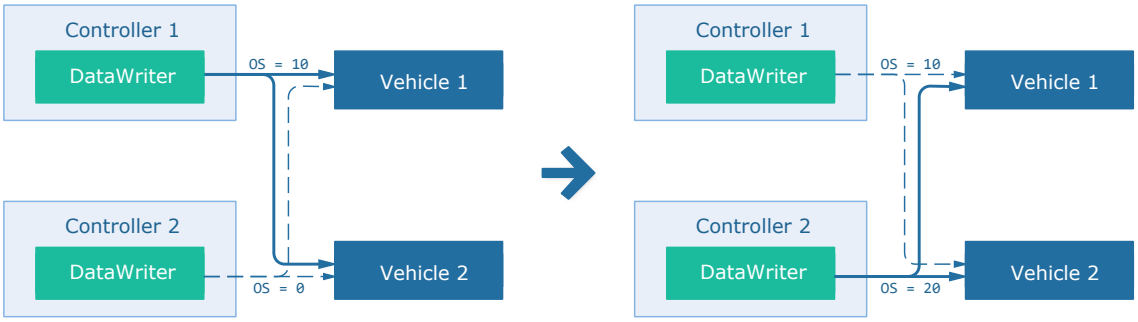

Figure 4.3: Problem caused by two controllers with only one data writer in each

to different vehicles at the same time. The priority writers are only created when active handover happens. The values of `OWNERSHIP_STRENGTH` of all the priority writers are the same and equal to a number larger than all the fail-safe writers' `OWNERSHIP_STRENGTH`. Besides, different from fail-safe writer, one priority writer can only write to a single vehicle. If the operator wants to seize control of several vehicles, multiple priority writers should be created. By introducing the priority writers, we do not need to change the value of any fail-safe writers, and the active handover will not influence other vehicles in the network.

For example, still consider the situation above, but priority writers are added, as shown in Figure 4.4. When Controller 2 wants to seize the control of Vehicle 2, Controller 2 only needs to create a priority writer to Vehicle 2. Vehicle 2 will then automatically be owned by the priority writer. At the same time, Vehicle 1 is still being controlled by Controller 1, because the priority writer of Controller 2 will not write to Vehicle 1 and the writer with highest `OWNERSHIP_STRENGTH` among all the data writers to Vehicle 1 is still the data writer in Controller 1.

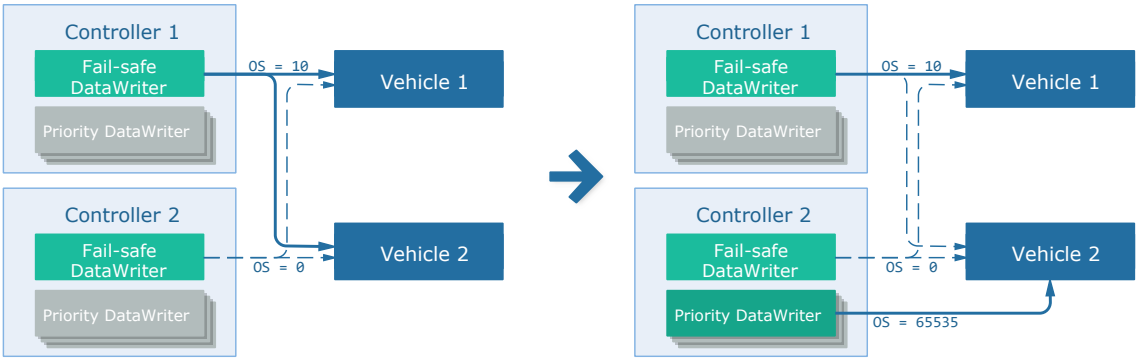

Figure 4.4: Controllers with fail-safe writer and priority writer

When Controller 1 wants to get the control of Vehicle 2 back, and Vehicle 2 is currently controlled by Controller 2 with a priority writer. The operator of Controller 1 can send a request to Controller 2. If the operator of Controller 2 approves this request, Controller 2 will immediately delete the priority writer to Vehicle 2, while Controller 1 will create the priority writer to Vehicle 2 after Controller 1 receives the approval message from the `RequestControl` topic. According to the handover scheme above, for one vehicle, there will be at most one priority writer in the network.

4.3.5 QoS Policy Summary

Based on the discussion above, we conclude the QoS for each topic as shown in following Table 4.2. The QoS policies that not listed in the table are all set to default values. These defalut values can be found in OpenDDS developer's guide (Object Computing, Inc., 2017b).

Notice the `OWNERSHIP_STRENGTH` of the fail-safe DataWriter in the vehicle is set to 0, since it is the safety layer in the operator control hierarchy. The `OWNERSHIP_STRENGTH` on controller is set according to the level in the operator control hierarchy. However, the largest value of fail-safe DataWriter's `OWNERSHIP_STRENGTH` should not exceed the `OWNERSHIP_STRENGTH` of the priority DataWriter. The `PRIORITY_OS` can be set to a number large enough, say 65535.

Table 4.2: QoS policy summary

<table>
<tr><th>Topic name</th><th colspan="5">QoS policies</th></tr>
<tr><td rowspan="3">DeviceInfo</td><td colspan="2">LIVELINESS.lease_duration</td><td colspan="3">DURABILITY</td></tr>
<tr><td>DataWriter</td><td>1s</td><td colspan="3" rowspan="2">TRANSIENT_LOCAL</td></tr>
<tr><td>DataReader</td><td>3s</td></tr>
<tr><td rowspan="5">ControlCommand</td><td colspan="2">LIVELINESS.lease_duration</td><td>OWNERSHIP</td><td colspan="2">OWNERSHIP_STRENGTH</td></tr>
<tr><td rowspan="2">Fail-safe DataWriter</td><td rowspan="3">1s</td><td rowspan="4">EXCLUSIVE</td><td>Vehicle</td><td>0</td></tr>
<tr><td>Controller</td><td>1 to PRIORITY_OS - 1</td></tr>
<tr><td>Priority DataWriter</td><td colspan="2">PRIORITY_OS</td></tr>
<tr><td>DataReader</td><td>3s</td><td colspan="2">–</td></tr>
<tr><td>RequestControl</td><td colspan="5">Default</td></tr>
</table>

4.3.6 Listing Ownership Relations

As it mentioned in the design of the `DeviceInfo` topic, the function of listing the relations of ownership between the controllers and vehicles needs to utilise `ControlCommand` topics. The handling of ownership is in the DDS implementation and the DDS does not provide the APIs for acquiring the ownership relations. We need to design a method to realise this function.

The designed method is that every time the control commands being written to the vehicle, if the `src_id` does not equal to the vehicle's `current_controller_id`, the vehicle updates `current_controller_id` field in its sample of `DeviceInfo` topic. The value will be changed to the `src_id` of the control command. Then the `DeviceInfo` writer will publish the sample in the network. All the devices in the network get this update on ownership relations. The reason for designing a method like this is that, if a data writer can write the control command to a vehicle, this data writer must be the owner of the data reader in the vehicle. The DDS guarantees this behaviour. Notice the vehicle only publishes new sample when the `current_controller_id` changes, therefore, the method will not cause a large amount of traffic during the `ControlCommand` topic being written by control devices.

As for the update timing, when there are new samples or liveliness changes in the `DeviceInfo` topic, every controller in the network that receives these messages will actively write commands to all the vehicles in the device list. The reason is that the new sample arriving and liveliness changing indicates that there are devices go online or offline, the ownership relations might be changed due to this. However, it is found in the preliminary experiments, and the source code of OpenDDS, the ownership information will not change until new samples write to the topic. Therefore, we need to probe the topic actively to acquire current ownership relations. Besides the new samples arriving and liveliness changing, the update will also happen after the active handover.

Also notice that the if the priority writer to one vehicle exists, only the priority writer will write to the vehicle. Otherwise, only the fail-safe writer will write to the vehicle.

The procedure of deriving the ownership relations is shown in Figure 4.5. The arrows in the figure mean the events will cause reactions in destination topic somehow other than the events write to the topic directly.

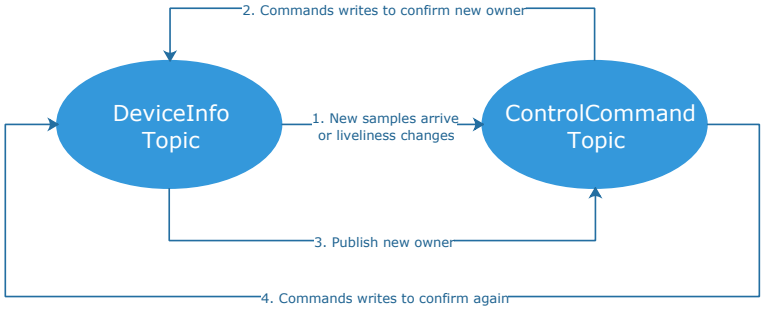

Figure 4.5: Procedure of updating relations

The trigger to update the relations is the new samples arriving or liveliness changing. The program on each controller will then write commands to all the vehicles in the device list. If the owner of the vehicle changes, the program in the vehicle will receive the control commands from the owner. If the `src_id` does not equal to the `current_controller_id`, the vehicle will publish a new sample in the `DeviceInfo` topic. The arriving of this new sample causes the fourth action in the figure to recheck the ownership. If there is no change in ownership relations, the update will end.

The updated ownership relations are in the `current_controller_id` field of each vehicle's `DeviceInfo` sample. Moreover, this sample has already published in the network, and every device has a copy of this relation. So, the operators can explicitly know the ownership relations between all the devices in the network.

4.4 Software Architecture

Based on the requirements and the functions, the software consists of three parts: the handover logic, the GUI application, and the vehicle application.

The handover logic is designed to handle the passive and active handover processes as it stated in previous section 4.3, and interact with the messages from the DDS. This part is reusable, which means the codes of the handover logic on the GUI application and the vehicle application is identical.

The GUI application is used to interact with the operators, and the vehicle application is used to receive the commands from the operators. Also, there is a safety checking in the vehicle application to ensure the commands that vehicles receive are valid and up-to-date.

To organise the codes in a flexible and maintainable way, we apply three-tier architecture to the software design. A three-tier architecture is a client-server software architecture pattern in which the user interface (presentation), functional process logic (business rules), computer data storage and data access are developed and maintained as independent modules (Eckerson, 1995).

In our case, the presentation layer refers to the GUI application for operators, also the vehicle application. We consider the DDS part as the data layer, which provides the functions of accessing and modifying data in the global data space. The logic layer refers to the handover logic, which handles the data from the DDS and takes corresponding actions.

The overview of the software architecture is shown in Figure 4.6. In the rest of this section, the detailed software architecture design will be given.

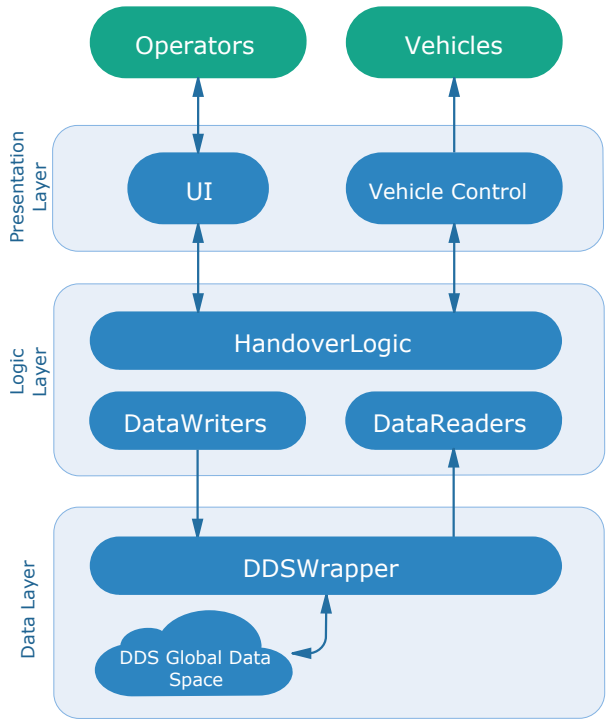

Figure 4.6: Overview of software architecture

4.4.1 Detailed Software Architecture Design

Android environment As it mentioned in section 3.3, the software should run on an Android tablet. This brings a problem, and that is, we cannot directly use OpenDDS APIs under Android environment.

The language widely used in Android application development is Java. According to the OpenDDS website (Object Computing, Inc., 2017c), OpenDDS is an open source C++ implementation of the DDS, and it supports Java bindings through Java Native Interface (JNI) (Oracle, 2017). However, the OpenDDS only supports for compiling under Android NDK (Object Computing, Inc., 2017a)but not Android SDK. The Java bindings are disabled under cross-compiling. This means we cannot use the Java bindings of OpenDDS under Android environment.

In the implementation, we write codes related to the DDS in C++ and use NDK to make the logic library running on an Android tablet.

Communications between the logic and GUI As it mentioned above, because of the differences in programming languages, we need to write the codes related to the DDS in C++ and the codes related to GUI in Java in the Android environment. This forces us to separate the GUI or vehicle application with the handover logic.

The separation increases reusability. When we want to build GUI or vehicle applications for another platform, we can leave the handover logic unchanged and only create the application for the specific platform. From the perspectives of the development time, this separation results in more time of development. However, this will give inspirations for designing and implementing the GUI applications with OpenDDS on different platforms.

JNI is the bridge between Java and C++. We can directly call the functions in the handover logic as shown in Figure 4.7

Nevertheless, there are drawbacks in using JNI.

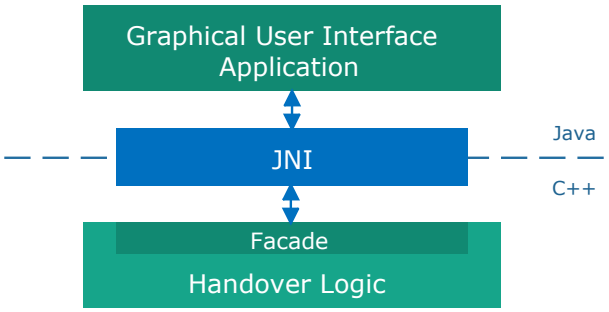

Figure 4.7: Interaction between C++ and Java codes via JNI

- It is hard to pass the parameters or results with the types other than the primitive types. JNI supports the arrays composed of single primitive type, such as integer arrays. However, we might need to transfer an array with customised data structures, e.g. `DeviceInfo`. Each instance of `DeviceInfo` contains three 64-bits integers and a string. This array cannot be transferred directly through JNI. Therefore, we need transfer the parameters via pointers. However, this adds the memory management issues, and it is error-prone in implementations.
- JNI only works for Java. In other programming languages, bridges between the languages and the C++ are different.

Consider these two drawbacks brought by JNI, we give up using JNI for communication, and we design another method to realise the communication between the logic codes written in C++ and the languages used in GUI development.

The new method is shown in Figure 4.8. The GUI applications do not call the logic directly. Instead, the GUI applications wrap the functions and parameters in a message packet and send the packet through the communication links to the process of logic codes. The process of logic codes will receive these messages from the communication links. Then, the process will handle these messages and take corresponding actions. This method decouples logic codes and GUI and makes it easier for extending in future development.

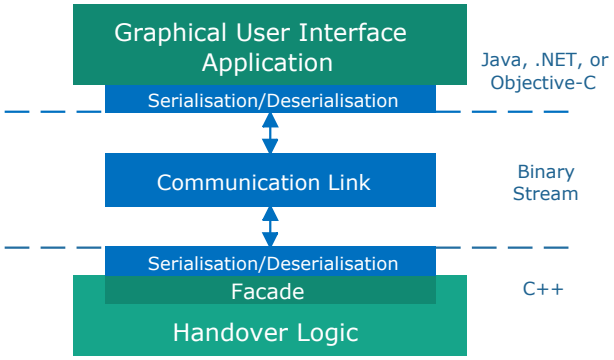

Figure 4.8: A method to communicate between C++ and Java codes without JNI

Serialisation As mentioned above, one of the inconveniences that JNI brings is the passing of customised data structures or objects between two languages. It is necessary for both the sender and receiver to recognise and translate the data structures into proper objects. Serialization helps us to do such actions.

Serialization is the process of translating data structures or object state into a format that can be stored or transmitted (for example, across a network connection link) and reconstructed later, possibly in a different computer environment. In our case, we refer the different computer environments to the different languages used in logic and GUI codes.

There are different types of serialisation methods. The serialisation and deserialization can be done manually. However, it is time-consuming to create this tailored solution since we need to write two sets of codes for both C++ language and the other language. Using third-party serialisation tools is an alternative. These tools are relatively mature. Some of the tools provide code-generation via interface description language files. This makes it easier to maintain and scale in future development.

The widespread serialization methods include Extensible Markup Language (XML), JavaScript Object Notation (JSON) and protocol buffers (Google Inc., 2017b), etc. The following texts will compare between these three methods.

- *XML*
  The initial goal of XML is to mark the documents on the Internet. Therefore, making it readable for both human and machine is embedded in its design concepts. However, when using in the serialisation, XML becomes verbose and complex.

- *JSON*
  Originated from the JavaScript, a weak type programming language, JSON uses the "Attribute-value" pair to describe the objects. JSON remains the human-readable feature. At the same time, JSON drastically reduces the size after serialisation compared to XML.

  The simplicity of JSON brings both advantages and disadvantages. During serialization and deserialization, there are no Interface description language (IDL) files used to constrain the design. If in the future, the development separated into business logic and GUI application development and assign these different development tasks to different developers, lack of uniform interfaces will cause inconvenience to develop and debug.

- *Protocol buffers*
  Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. Compared with XML, protocol buffers are simpler, 3 to 10 times smaller, are 20 to 100 times faster (Google Inc., 2017b). And protocol buffers use standard IDL and provide compiler for code-generation, therefore, protocol buffers are less ambiguous compared to both XML and JSON. Compared to JSON, the standard IDL of protocol buffers also ensure the scalability for future development.

According to the discussions above, due to the considerations of reusability and scalability, we choose protocol buffers as the serialisation tool.

Communication links Since we give up the scheme of directly calling APIs from the handover logic, communication mechanisms should be introduced to make different languages communicate with each other.

For the development upon the Android layer, Google provides Android-specific IPC mechanisms, primarily including Intents, Binder, and Messenger (Shao et al., 2016). For the development upon native layer, as Android also relies upon a tailored Linux environment, it provides a subset of traditional Linux IPCs (which are distinct from Android IPCs), such as signals, network sockets, and Unix domain sockets.

Despite there are various mechanisms provided by Android or Linux, the IPC mechanism that can be used in both the Android Java and native layer is limited to the Unix domain socket.

The Android IPCs are not suited to support communications between an application's Java and native processes or threads. While there are Android IPC APIs available in API, no such API exists in the native layer. To realise cross-layer IPC, we can only employ Unix domain sockets.

Although the choice of Unix domain socket seems to be a compromised solution, it can also bring advantages to reusability. The Unix domain socket facility is a standard component of POSIX operating systems. Furthermore, the Unix domain sockets are similar to network sockets. Almost all languages and frameworks provide the supports for socket communication. The Unix domain sockets can be easily transformed to network sockets if the languages or platforms used do not support for Unix domain sockets. Therefore, the Unix domain sockets can be widely used in different platforms.

Vehicle applications The main jobs of vehicle applications include receiving commands from the DDS, translating the commands to the vehicle-specific controlling signals and sending these signals to the servo controllers in the vehicles.

As it mentioned in section 2.3, the embedded computers in the vehicles are RaMstix and Raspberry Pi. Both of them run Linux. Distinct from the GUI application, the vehicle application is capable run code under C++. In the development of the vehicle applications, since both the applications and the handover logic are written in C++, the vehicle applications could directly call the functions in handover logic without any conversion. However, we design that the applications that control the vehicle, and the handover logic related to the DDS are running in different processes; the applications still need to communicate with the handover logic via IPC. The reason is that we want to increase the robustness of the vehicle applications. The applications can still run under the situation where the process of the handover logic crashes. In this situation, the application will send signals to the servo controllers to make the vehicles remain stable to prevent unexpected behaviours of the vehicles.

To reuse the codes as much as possible, we use Unix domain socket mechanism as we use in the GUI development.

Based on the discussions above, a detailed architecture can be derived as shown in Figure 4.9.

4.5 Software Design

In this section, the detailed design of the software will be given. As it mentioned in the last section, we separate the software architecture into the handover library, the GUI application and the vehicle application. We follow the order in this section as well.

To show a general software structure, we give the package diagram in Figure 4.10.

We divide the software into three layers, and each layer only depends on the layer below it. This benefits the decoupling and makes the software clearer. In the following texts of this section, we give the software design in different packages.

4.5.1 DDS Package

This package is designed to wrap the functions in the OpenDDS library. This package hides the details of building up DDS entities for the clients. The clients only need to provide initialization parameters to the `DDSWrapper` and the `DDSWrapper` will build the DDS entities, e.g. publishers, subscribers, topics for the clients. When the clients need these entities, they could just get the entities from `DDSWrapper`.

The class diagram of the DDS package is shown as Figure 4.11

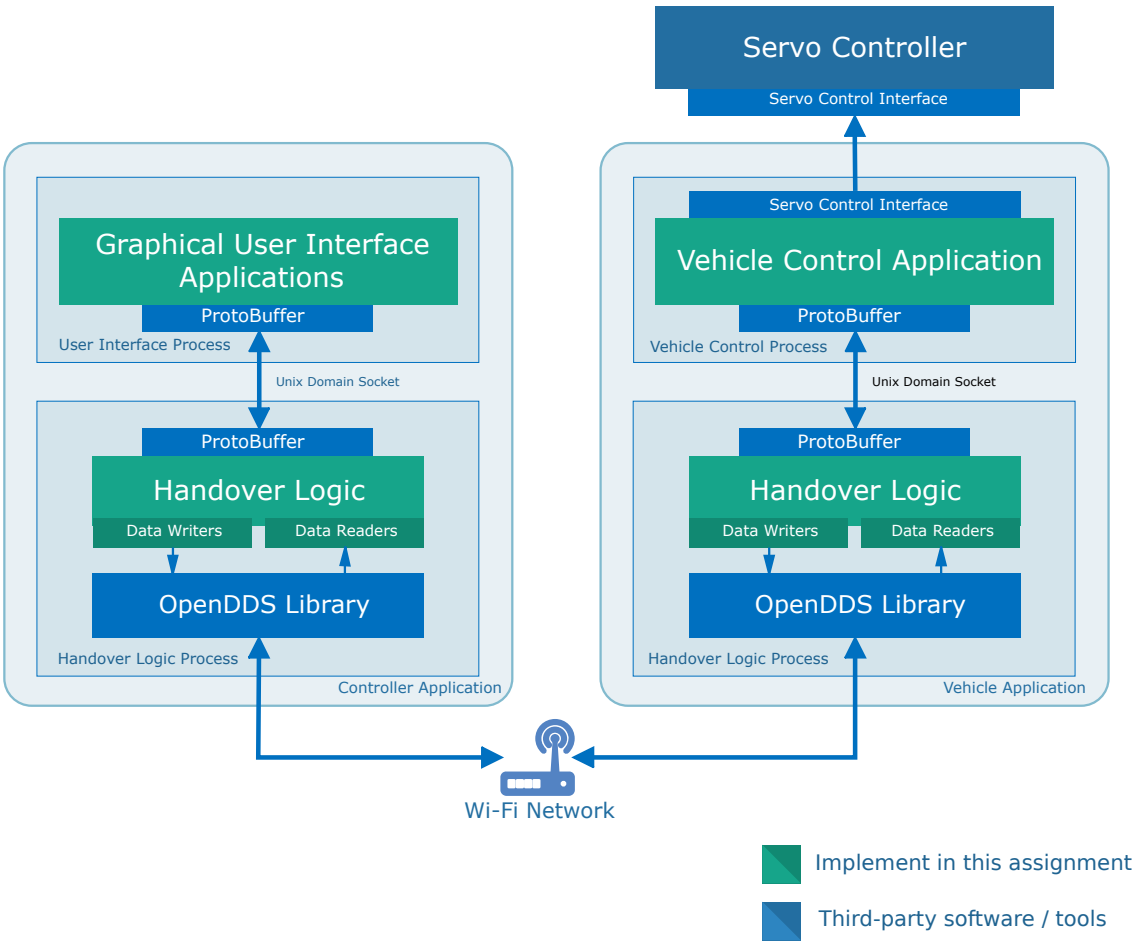

Figure 4.9: Detailed architecture

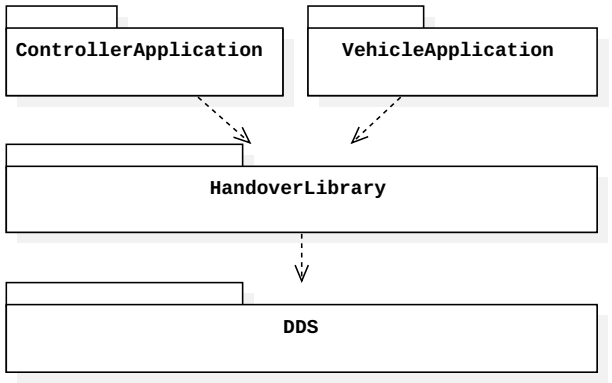

Figure 4.10: Software package diagram

4.5.2 Handover Library

The handover library contains the logic code for the handover process. Moreover, for the controller and vehicle application, these logic codes are almost identical. The functions of the handover library include:

- handling data from the DDS layer,
- sending data to the DDS layer to publish information to the network,

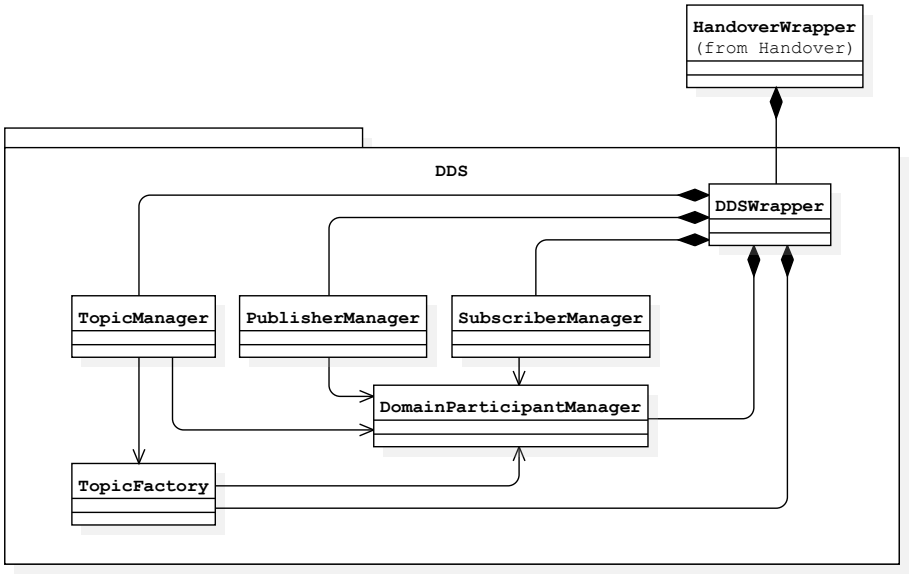

Figure 4.11: DDS package class diagram

- providing interfaces for the core functions of the logic codes.

According to the functions listing above, we design classes as shown in Figure 4.12.

DDS data access The data writers and data readers are recognised as the interface with the data layer. We create the `DataWriterManager` and `DataReaderManager` to handle the logic of data writing and reading.

For the clients, it is convenient to be provided with a uniform interface for writing data. For each topic, there is a specific data type. Therefore, we could use the static polymorphism in C++. The clients just need to write one data type of message, the function overloading of C++ will determine which function needs to be evoked. The `DataWriterManager` is used to realise these features.

Handle passive handover logic The `DeviceInfoManager`, `ControlCommandManager` and `RequestControlManager` are used to manage the logic of handover. Discussed in section 4.3, the passive handover process need the collaborations between different topic. Thus, we give the diagrams that show the procedure of both passive and active handover as shown in Figure 4.13

As is shown in Figure 4.13, the start point of passive handover is the changing of the liveliness of the `DeviceInfo` topic. This indicates that there are devices go online or offline. The ownership of the vehicle can be changed due to this. Thus, every controller in the network needs to confirm if it is the new owner of the vehicle. Therefore, every controller writes a probe control command to all the vehicles in the local device list.

If the ownership of the vehicle changes, the vehicle can only receive the control commands from the new owner. If the source ID of the control command message is different from the current controller ID in the `DeviceInfoManager`, the vehicle will publish a `DeviceInfo` sample with the new controller ID. This informs all the devices in the network that this vehicle's owner has changed. Then the controller will send a probe control command again, however, since the owner of the vehicle is identical to the sender, the vehicle will not publish any `DeviceInfo` samples this time. The passive handover ends here.

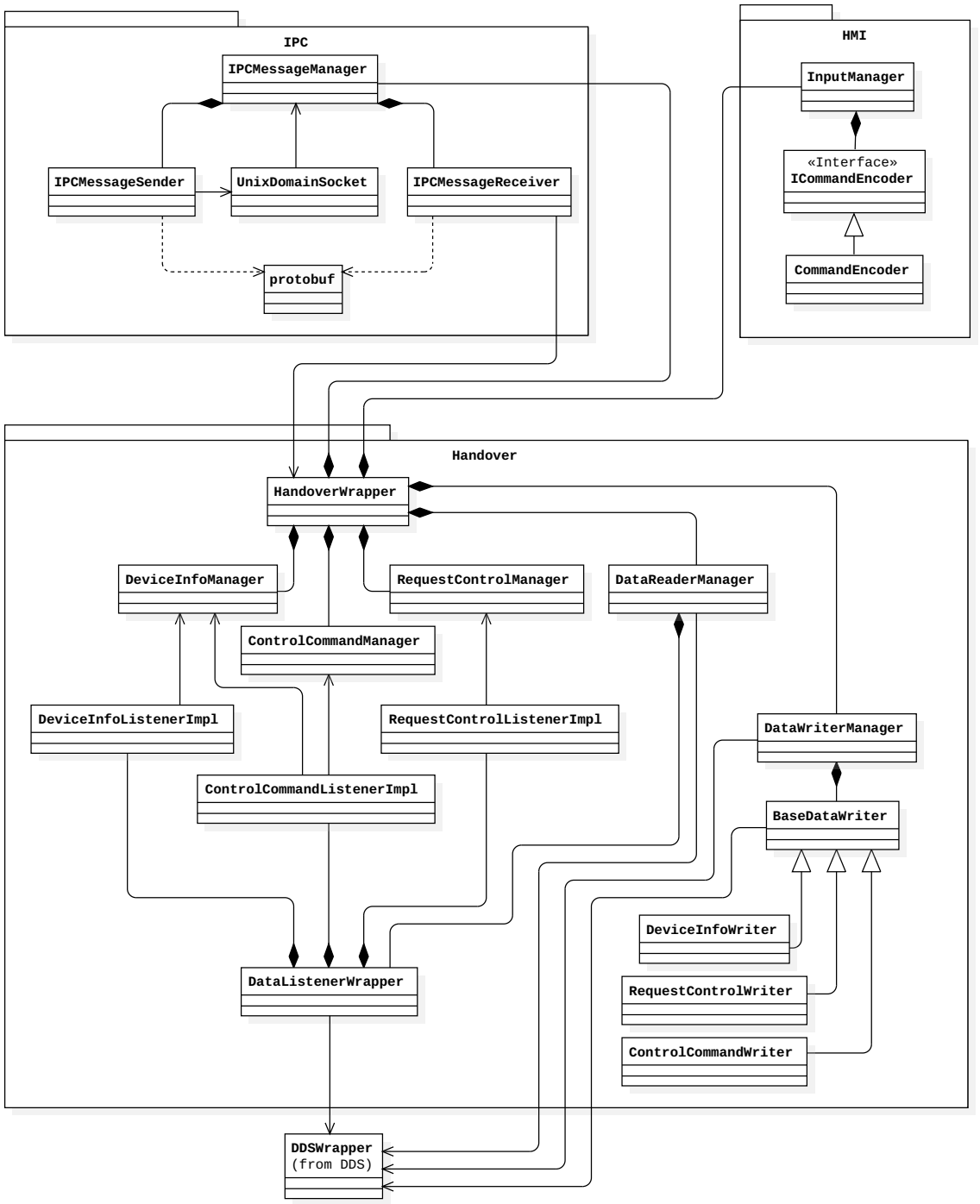

Figure 4.12: HandoverLibrary package class diagram

Handle active handover logic The active handover needs the collaborations between all the topics, and an activity diagram showing the procedure of the active handover is shown in Figure 4.14

As is shown in Figure 4.14, two characters in the active handover process is the requester and the replier. The handover process is listed as below.

1. *Requester sends request.* The requester sends a request to a selected vehicle. The requester needs to fill four fields in a `RequestControl` sample: the ID of the selected vehicle, the ID of the replier, and the ID of requester itself. The operator specifies the

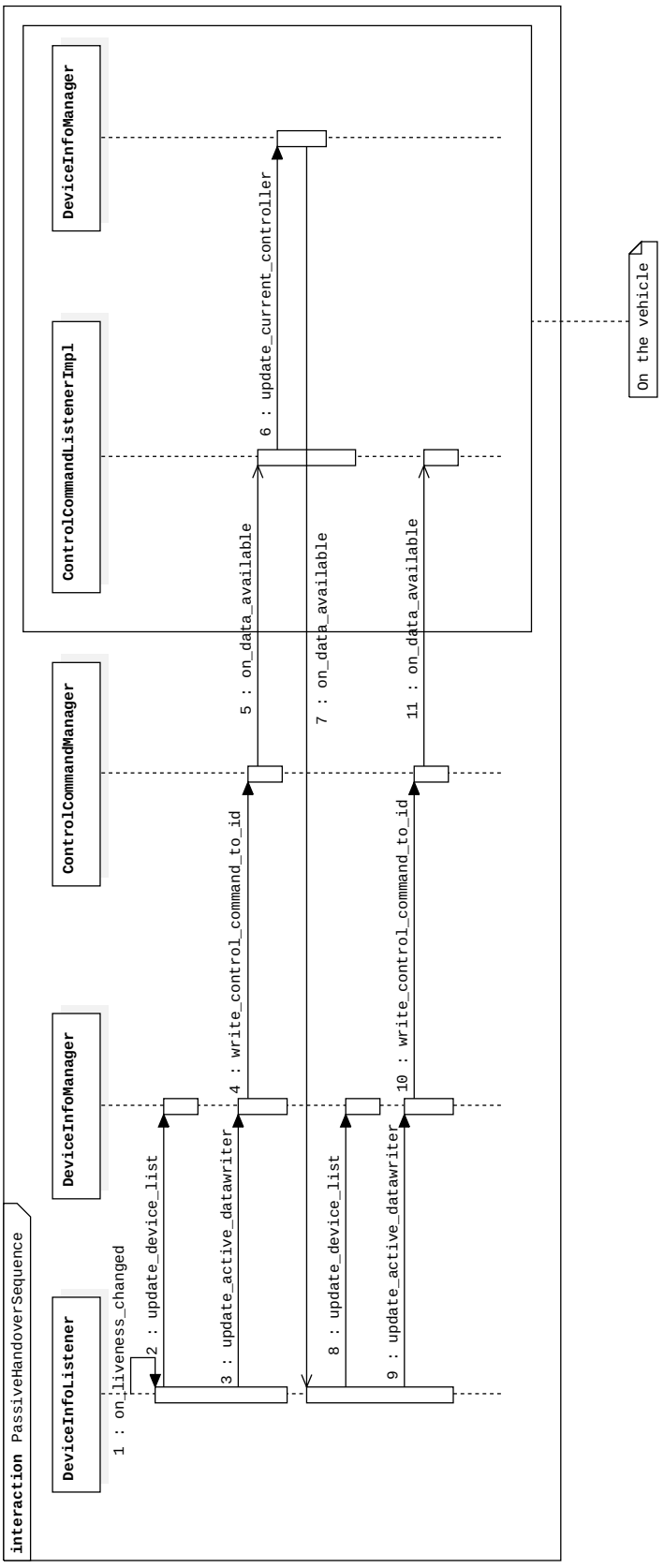

Figure 4.13: Passive handover sequence diagram

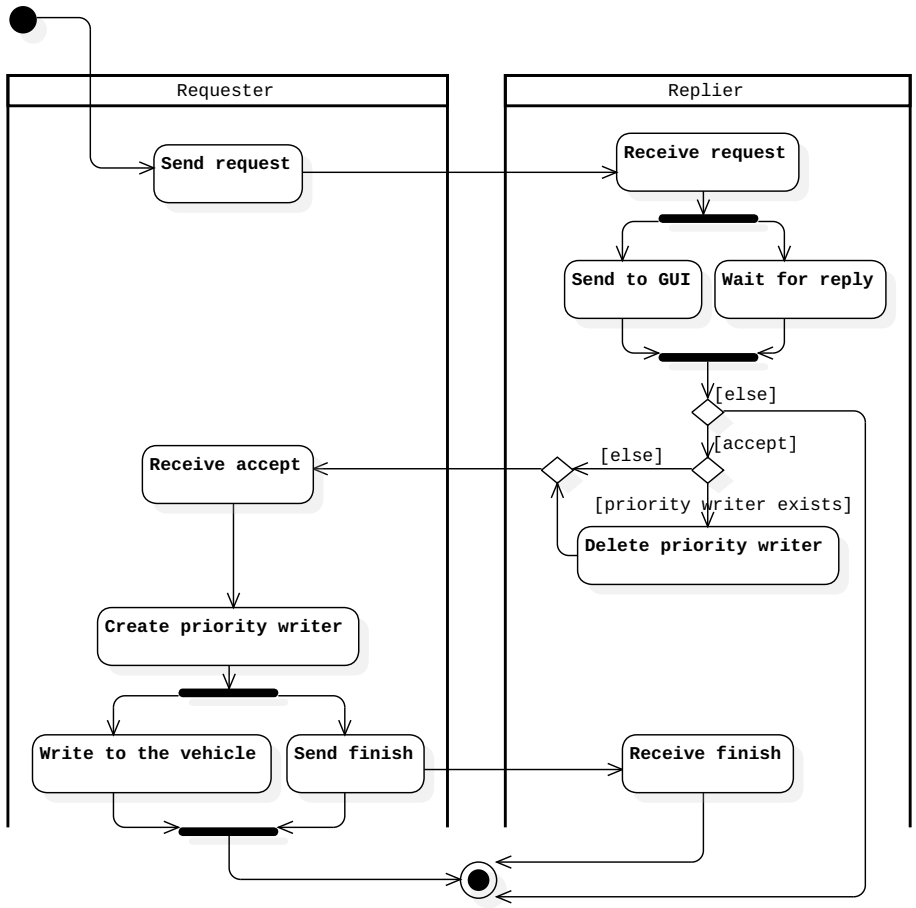

Figure 4.14: Active handover activity diagram

vehicle ID. The ID of the replier and the ID of the requester can be derived by checking the local device list in `DeviceInfoManager`.

2. *Replier receives the request.* Upon the replier receives the request, it will send a notification to the GUI to inform the operator that there is a request from the network. If the operator approves this request, the replier will first check whether the priority writer to the vehicle ID exists. If so, the replier will delete this priority writer. Then, the replier will modify the `ack` field to `ACCEPT` and publish this reply in the network.

3. *Requester receives the approval message.* The requester will create a priority writer to the vehicle, and immediately write a control command to the vehicle to update the ownership relation. Then the requester will modify the `ack` field to `FINISH` to replier to indicate the handover process is finished.

Interface with the GUI application The `HandoverWrapper` is a facade class to provide the interface of handover logic. Also, the `HandoverWrapper` is responsible for creating the different objects and hiding the details for the clients.

As for the communication, we design the `IPCMessageManager` to wrap the functions of protocol buffers and Unix domain socket. The GUI application can send protocol buffers messages to the Unix domain socket, and the `IPCMessageManager` running in the `HandoverLibrary` process will decode the message and take corresponding actions.

Translate user inputs to the control data model The class `InputManager` is used to convert user inputs to the control data model. The converting function can be put in the GUI application, however, consider the reusability of the `HandoverLibrary`, we put the converting function in the `HandoverLibrary`. Therefore, the GUI application is only responsible for receiving user inputs and forwarding these inputs to the `HandoverLibrary`.

4.5.3 GUI Application on Android

In this subsection, the design of the GUI application on Android is discussed. We first choose the GUI pattern and then we will give the class diagram of the GUI application.

GUI architectural pattern In the GUI development, there are some solutions for general challenges, and we refer these solutions to as patterns. In Android development, there are three popular design patterns or architectural patterns, Model-View-Controller (MVC) pattern, Model-View-Presenter (MVP) pattern, and Model-View-ViewModel (MVVM) pattern. In the design, because the scale of the application is relatively small, the MVVM pattern brings a large amount of learning costs, we abandon the MVVM pattern and choose between the MVC, and MVP pattern.

- MVC
  MVC is a proven and widespread pattern. It separates the roles of an application into Model, View and Controller, where Model is responsible for the business logic, View is the user interface, and Controller is the mediator between the View and the Model. The interactions of these three parts are shown in Figure 4.15.

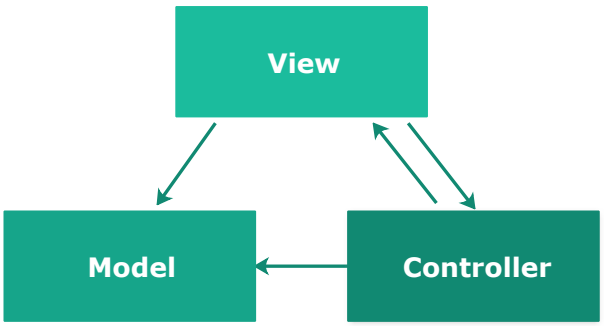

Figure 4.15: Interactions between Model, View and Controller

  However, when we put MVC into the context of Android development, the ambiguity between View and Controller make the Controller (business logic) completely tied to View role (Activity/Fragment). This increases the complexity of the codes and brings inconvenience for unit testing.

- MVP
  The MVP pattern was developed using the same principle as the MVC pattern, separation of concerns. However, compared to the MVC pattern, the MVP uses modern paradigm and creates a better separation of concerns.

  The MVP pattern separates the application into Model, View and Presenter. The Model holds the business logic of the application. The View passively updates the data and forwards the user operations to the Presenter. The Presenter works as the mediator. It retrieves data from the Model and handle the users' operations forwarded from the View. The interactions between each part are shown in Figure 4.16.

  Different from the MVC pattern, the MVP pattern cuts the paths from the View to the Model, and the Presenter mediates all the communications between the Model and View.

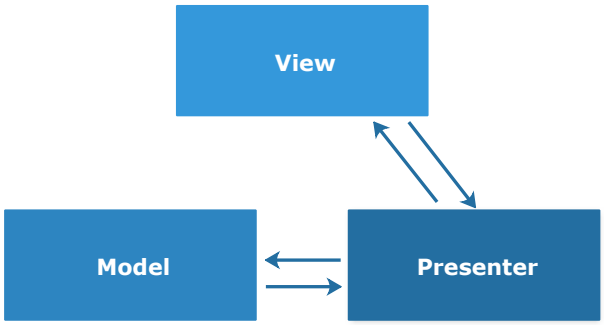

Figure 4.16: Interactions between Model, View and Presenter

The View updates passively. Besides, one Presenter binds to a single View. This benefits the separations of concerns in three layers.

Consider the discussions above, we choose to use the MVP pattern to decouple the View and the Model in the application. Using the MVP pattern makes the structure clearer and easy to understand.

Software design As it discussed above, we choose the MVP pattern as the architectural pattern in the GUI application development. We design the software in this pattern as follows.

- *Model*. The APIs provided by the `HandoverLibrary` can be recognized as the data layer for the GUI. Firstly, the Model in the application is designed to wrap the requests to the protocol buffer packet and send to `HandoverLibrary` process via Unix domain socket. Secondly, the Model also receives packets from `HandoverLibrary` process and parse the packets from binary streams, and forwards the parsed objects to the Presenter.

- *Presenter*. The Presenter is the mediator between the Model and the View. The Presenter receives information from the Model. The messages sent from the Model include the device lists and the control requests from other controllers. The Presenter handles these messages and evokes the View to update the user interface.

  The Presenter also processes user operations from the View. These operations are joystick inputs, operators' control requests to the vehicles, and the replies of the operators to the requests from other controllers. The Presenter handles these messages and calls corresponding functions in the Model.

- *View*. The View is used to display current devices in the network and receive user inputs. The View does not actively update the user interface. The View only passively waits for the Presenter to update the user interface.

Considering the functions of each character above, we design the classes of the GUI application as shown in Figure 4.17.

Running HandoverLibrary process As it stated in subsection 4.4.1, we design that the GUI application running in one process and `HandoverLibrary` running in another process. To realise this, we could directly create the `HandoverWrapper` object in the `MainActivity`. However, the `MainActivity` thread in Android is responsible for the UI rendering, running `HandoverWrapper` in `MainActivity` will slow down the response speed of the UI operations. Therefore, we utilise the Service application component (Google Inc., 2017c) in Android to make the `HandoverLibrary` running in the background. The Service runs in a different

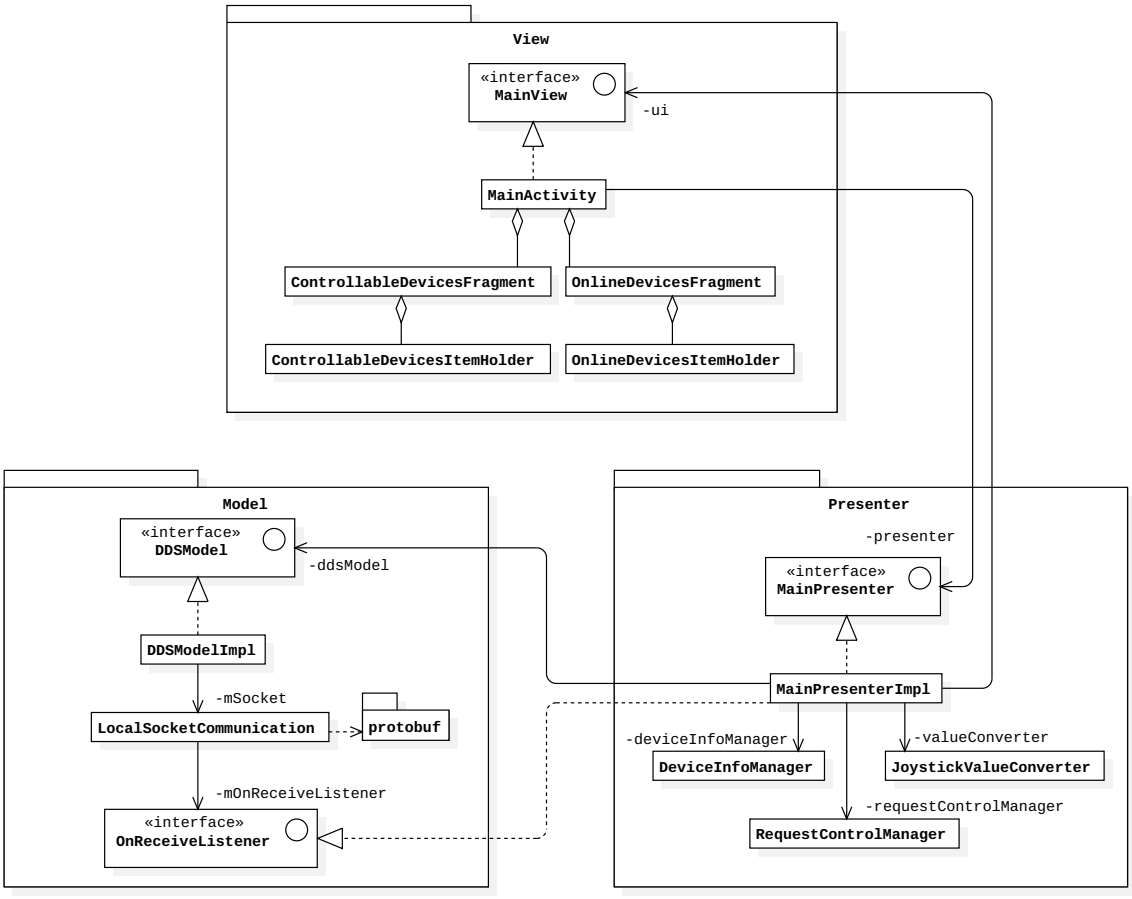

Figure 4.17: Class diagram of Android GUI application

process from the `MainActivity`. Thus, the `HandoverWrapper` object is established in another process, and the `HandoverWrapper` object will not slow down the UI.

4.5.4 Vehicle Application

The main functions of the vehicle application include:

- receiving control commands from the `HandoverLibrary` process,
- translating the control commands to the data structure that can be recognized by the servo controllers in the vehicle, and write to the servo controllers,
- running a safety checking thread that if the program does not receive control commands from the `HandoverLibrary` process for a certain time, the vehicle program will write a stop or hover command to the servo controllers onboard.

Classes are designed to realise such functions. The class diagram of the vehicle application is shown in Figure 4.18. We separate the variables in different vehicles. There are two main variables for different vehicles, the data structures of control signals and the hardware interfaces between the servo controllers and the embedded computers. Currently, there are two types of vehicles presented in RoVe, Eyedrive rovers and DJI Phantom 3 drones and two types of embedded computer boards, RaMstix and Raspberry Pi. Thus, we designed `ICommandDecoder` and `IOnBoardController` to separate these variables as shown in the figure.

The `SafetyChecker` runs a watchdog. When the watchdog does not receive any commands from the `HandoverLibrary` process for a certain period, the `SafetyChecker` will write

stop or hover commands to the vehicle to prevent unexpected behaviours. The control commands write to the vehicle at a frequency of 200 Hz. We set the period to 20 intervals, i.e. 100 milliseconds, which means the `SafetyChecker` stops or hovers the vehicles if it does not receive any command during last 100 milliseconds.

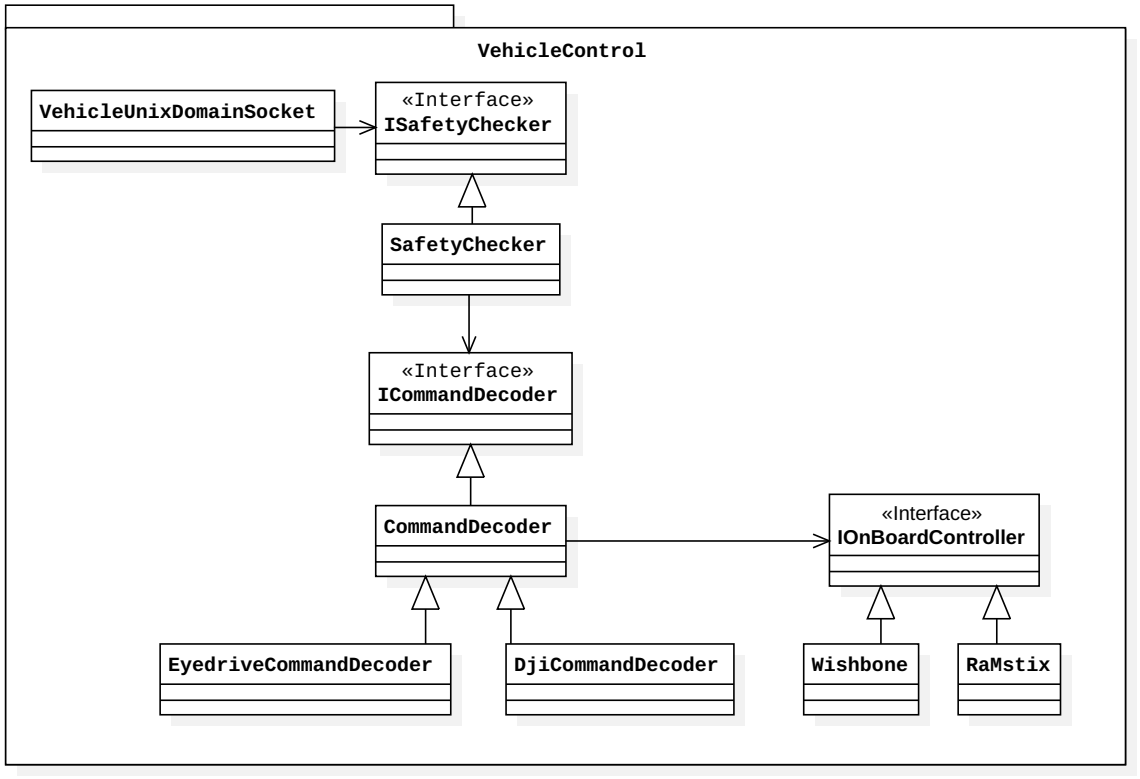

Figure 4.18: Class diagram of vehicle application

5 Results

The previous chapter introduces the hierarchy and the software design. The performance of the handover process is discussed in this chapter since the primary function of the operator control hierarchy is the handover of the control. We conduct experiments to measure the response time of both the passive and active handover process. At the same time, the experiments simulating multiple devices being in the network are performed to predict the influence of device increasing in the network. The experiments are also preformed to test the correctness of the control and command module. The experiments show that the operator can use a joystick to control movements of an Eyedrive rover.

5.1 Testing Setup

To test both the passive and active handover process, we use the testing setup as shown in Figure 5.1. The handover process is between two devices. The handover of control can be from one controller to another controller or from controller to the vehicle. Therefore, we use a laptop as the controller and Eyedrive with Raspberry Pi from the project RoVe as the vehicle in the testing setup. The Android tablet is not used as the controller because it is hard to automate the tests on Android. The behaviours of the handover should be identical to the laptop except for the platform performance, since the Android program uses the same code as the Linux program running on the laptop.

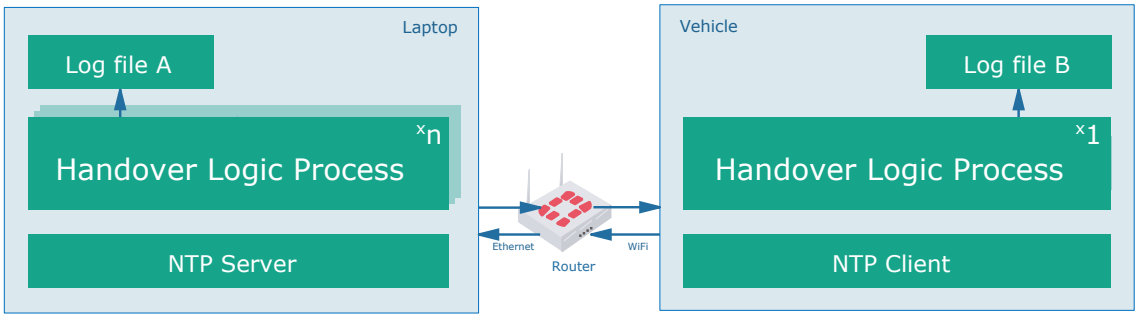

Figure 5.1: Schematic overview measurement setup

We simulate multiple control devices by running different processes of handover logic on the same machine, as shown in the figure. Logging files are used to store timestamps. The time on the laptop and the vehicle are synchronized by the Network Time Protocol (NTP)[1].

5.2 Measurements

The performance of the handover process is measured according to the response time of the handover process. The response time is the time between one controller getting offline to another controller taking control of the vehicle. The first timestamp is recorded at the time of program quitting. The second timestamp is recorded when the vehicle publishes a DeviceInfo sample with a new `current_controller_id`.

5.2.1 Passive Handover

Passive handover happens when new devices join the network, or current devices quit the network. To get notifications of these events, we utilise `Liveliness` QoS policy in DDS. Availability of one controller is determined by the liveliness of the data writers in the controller. The

[1] http://www.ntp.org/

middleware generates liveliness changing signals when devices join or quit. We capture these signal changes, and probe all the vehicles in the network to confirm the ownership relations.

The DDS middleware broadcasts a message when one DDS program ends normally. All other DDS programs in the network receive this message and the liveliness changing signals. However, when the program crashes without shutting down the DDS services, or the program temporarily drops out of the network because of the network interruptions, other programs in the network will not immediately receive the messages indicating there are devices offline. The program will only get the notifications when the liveliness assertion in data readers happens.

Therefore, in the testing of passive handover, two experiments are conducted. The first one is aiming to simulate the situation where the controller program ends normally. The second one is the situation where the controller program crashes. The latter situation can also simulate the handover behaviours under network interruptions because both program crashing and network interruptions are detected during the data readers'liveliness assertions.

To measure the performance under normal quitting, we run the program and then end the program with "`return 0`" to simulate a normal quitting. The cleanup of DDS entities will be called in the destructor of the `HandoverWrapper`. Then we record the timestamp of the program quitting and the timestamp of the `current_controller_id` of the vehicle changing. The response time is the duration between these two timestamps.

To measure the performance under program crashing, we manually call "`abort()`" in the main program to make the program crash. Additionally, we set `duration_time` in `Liveliness` QoS policy to different values and run the tests to compare the response time under different `Liveliness` settings.

5.2.2 Active Handover

Active handover happens when the operators actively send requests to another controller. As it mentioned in previous subsection 4.3.3, the procedure of active handover includes the requesters sending requests, the repliers replying to these requests, then both the requesters and the repliers taking actions to handover the control. We measure the response time from the repliers publishing accepting messages to the vehicles publishing new `current_controller_id` in the network. To simulate the multiple devices situation, we run different numbers of controllers process.

To exclude the time waiting for the operators'permissions, we modify the code in the `HandoverLibrary`. In the test, the program will directly accept requests, instead of asking permissions from the operators after receiving the requests.

5.3 Results and Analysis

5.3.1 Passive Handover under Program Normally Quitting

The result of the passive handover experiment on normal quitting is shown in Table 5.1. Moreover, a box plot diagram showing the comparisons of different numbers of control devices is given in Figure 5.2.

We start the process and end it normally after 10 s, recording the response time between the process quitting and the `DeviceInfo` of the vehicle changing. For each number of controllers, we repeat the test for 100 times and calculate the result. The result is shown in Table 5.1.

The number of controllers indicates how many controller processes are created on the Linux machine. The first column of the result shows that only one controller is on the network. Therefore, the handover is between the controller and the vehicle onboard controller. The second column shows that two controllers are in the network. We make one of the controllers start and end periodically as the previous test, and make another controller remain online in the

network. The third and fourth column are the same: one of the controllers starts and ends periodically, and the number of the controllers that remain online increases to two and three respectively.

As is shown in the table, when the number of controllers increases, the average time of the passive handover has an upward tendency. Moreover, the standard deviation also raises. The reason might be that the DDS middleware needs more network packets and computation time to derive the availability of the remote data writers and data readers.

Table 5.1: Response time of passive handover under program normally quitting

Number of controllers	**1**	**1+1**	**2+1**	**3+1**
Average (ms)	31.8	38.6	37.8	50.4
Stdev (ms)	7.5	9.5	9.5	12.3
Min (ms)	18.6	25.1	27.3	31.7
Max (ms)	59.3	78.7	77.4	96.2

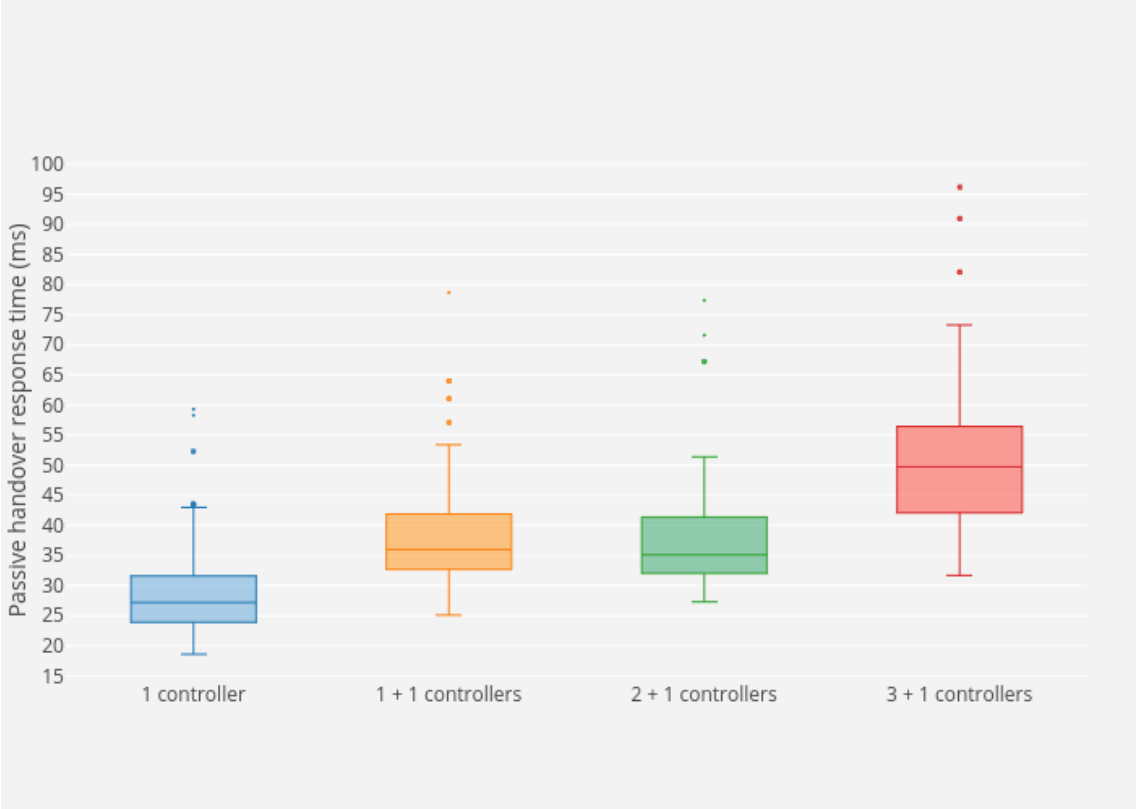

Figure 5.2: Box plot of passive handover response time

5.3.2 Passive Handover under Program Crashing

The result of the passive handover experiment on program crashing is shown in Table 5.2. We run controller process and crash the process after 10 s. Then we record the time between program crashing and the `DeviceInfo` sample of the vehicle changing. The test is repeated 100 times for each `lease_duration` setting.

As is shown in the table, the time of passive handover depends on the liveliness assertion time, which is to be expected. The clients can determine the specific value of `lease_duration` based on their needs. Notice that the smaller value of `lease_duration` brings more network

overheads. The influence of these overheads will become significant when there are multiple publishers in the network.

Table 5.2: Response time of passive handover with program crashing

Liveliness assertion intervals (s)	**0.5**	**1**	**3**
Average (ms)	424.4	693.1	2701.4
Stdev (ms)	22.5	18.0	17.9
Min (ms)	389.7	634.1	2638.2
Max (ms)	478.3	768.2	2771.2

5.3.3 Active Handover

The result of the active handover experiment is shown in Table 5.3. The number of controllers indicates how many controllers were in the network. The active handover is between all of the controllers. The test for each number of controllers is conducted 200 times.

The average response time of active handover is independent of the number of controllers in the network, which is different from the passive handover. The reason is that only two publishers, the requester and the replier, are involved in the active handover.

Notice that the standard deviations are relatively large. Test shows that sometimes the ping-pong handover happens. For instance, after Controller A successfully seizes the control of the vehicle from Controller B, Controller B can still write to the vehicle, despite Controller B deletes its priority writer. This is unexpected and results in that the control falls back to Controller B. However, the changing of the current controller in the vehicle triggers Controller A to write to the vehicle again, and then Controller A seizes again. After this, the control hands over to Controller A and will not fall back to the Controller B for a second time. The occurrences of this ping-pong handover are random and cause the large standard deviations in response time.

Table 5.3: Response time of active handover

Number of controllers	**2**	**3**	**4**
Average (ms)	23.7	22.5	24.1
Stdev (ms)	16.9	17.6	15.9
Min (ms)	4.8	4.4	2.5
Max (ms)	120.5	150.4	63.1

6 Conclusions and Recommendations

In this chapter, We conclude the contributions of this assignment. Then, we revisit the requirements in section 3.3 to evaluate the completion of the requirements. Finally, recommendations are given for future work on the hierarchy and software development based on DDS middleware.

6.1 Conclusions

The goal of this assignment is designing an operator control hierarchy for multiple robot systems, designing a control command module, implementing the hierarchy and giving a demonstration on real setups, and testing the correctness and efficiency of the hierarchy.

In this report, we introduce the background and analyse the requirements for the assignment. Then, the design and implementation of the operator control hierarchy are elaborated. Finally, the tests are conducted. The tests show that the hierarchy worked as expected, and temporal behaviours of the hierarchy are concluded according to the tests.

6.2 Requirement Evaluation

Hierarchy

- We design the software based on the DDS standards and maximumly utilise the Quality of Service in the DDS, as is shown in the design of topics in section 4.3. This satisfies Requirement 1.
- To handle the temporary control requests in the common usage, we create the `RequestControl` topic and add the priority writers to the `ControlCommand` topic to handle these requests, as it stated in subsection 4.3.3. This satisfies Requirement 2.

Control and command module

- In section 4.2, we design the control data model by abstracting the input signals used by different autopilots, so that this control data model can control different types of vehicles. These designs satisfy Requirement 3.
- We design both input and output adapter in subsection 4.5.2 to take inputs from different input devices and output the movement commands to the servo controller, which satisfies Requirement 4 and 5.

Implementation

- We implement the software based on OpenDDS library, which satisfies Requirement 6.
- We design and implement the GUI application on both a Linux laptop and an Android tablet. This satisfies Requirement 6, 7, and 8.
- In the implementation, we separate the business logic and user interface. This makes the software structure clearer. The code related to business logic are written in C++, which ensures the portability among different platforms. The user interface is implemented using Java on an Android tablet, and Qt is used to build user interface on a Linux laptop. We design a method to communicate between the business logic and the user interface. This approach enables developing the business logic and the user interface in different languages.

Tests

- To examine the performance of the handover process of the hierarchy, we conduct experiments as it stated in Chapter 5. The first test of passive handover shows the correctness of the passive handover, which satisfies Requirement 10.
- Since we design that the controllers and vehicles run the identical code, the laptop can be easily simulated as a vehicle in the network. This satisfies Requirement 13.
- We test the control data model only on Eyedrive with Raspberry Pi. Due to the limited time, the control data model on other vehicles has not been tested yet. Therefore, Requirement 12 has not been fully achieved.

General requirements

- In the software design and implementation, we take the reusability, scalability, and extensibility into consideration. We design the method to communicate between different programming languages. The method can be easily port to a new language. Moreover, we design a package to wrap the DDS functions, which can also be reused in the future development. We design the software following the 3-tier architecture and apply the MVP pattern to the implementation of Android GUI application. Therefore, the software can be easily extended. Thus, Requirement 15 is achieved.
- During the implementation, we utilise protocol buffers and C++11 features to simplify the programming, and polymorphism is frequently used to reduce the repeating of code, which satisfies Requirement 16.

6.3 Recommendations

Based on the conclusions and experience of OpenDDS, several recommendations are given:

Testing the control data model Due to the time constraints, we only implemented the output adapter of the control data model for Eyedrive with Raspberry Pi. The code for DJI Phantom 3 with Raspberry Pi has been written but without testing yet. The interface for the RaMstix has also been done but without testing. The implementations for these platforms can be done in the future.

Optimizing the device listing function The online device listing function is implement based on the `Durability` QoS of DDS. However, it is found in the tests, sometimes, for later-joining subscribers, the OpenDDS does not send historical samples in the `DeviceInfo` topic immediately, but after about 60 s. This issue appears more frequently if we run more than three programs in the network. For some circumstances, this could be unbearable. However, we have not found a solution by simply tweaking the QoS of the DDS participants related to `DeviceInfo` topic yet. We have tried to update the `DeviceInfo` samples periodically but this is an inelegant solution since it increases the traffic in the network. A new method should be designed to tackle this problem in the future.

Testing temporal behaviours on real platforms Due to the limitation of time, we conduct experiments on a single machine to predict the behaviours under multiple operators and vehicles. However, the tests on the real platforms still need to be done in the future.

A Using OpenDDS in Android

In this appendix, we introduce the building of OpenDDS library for Android, along with the usage of the library in Android.

A.1 Building OpenDDS Library for Android

1. Download and setup Android NDK. Follow Google Inc. (2017d) to make standalone toolchain.

2. Add path to the `$PATH_TO_TOOLCHAIN/bin` to the environment variables.

3. Add following contents to the `configure` file under OpenDDS source folder (in the "my %platforminfo" section) :

```
'aarch64' => {
    'libpath' => 'LD_LIBRARY_PATH',
    'no_host' => 1,
    'aceplatform' => 'android',
    'aceconfig' => 'android',
},
```

4. Run in the terminal:

```
./configure --target=aarch64 --target-compiler=aarch64-linux-
    android- --no-test --static
```

5. Open

```
$OPENDDS_ROOT/build/target/ACE_wrappers/include/makeinclude/
    platform_macros.GNU
```

   change last 3 lines to:

```
CC = aarch64-linux-android-g++ -pie -fPIE
CXX = aarch64-linux-android-g++ -pie -fPIE
LD = aarch64-linux-android-g++ -pie -fPIE
```

6. Open

```
$OPENDDS_ROOT/build/target/ACE_wrappers/ace/config-android.h
```

   comment out "`# error Unsupported Android release`" in line 374.

7. Open

```
$OPENDDS_ROOT/build/target/ACE_wrappers/ace/Basic_Types.h
```

   add following text after line 380:

```
#define ACE_LITTLE_ENDIAN 0x0123
#define ACE_BYTE_ORDER ACE_LITTLE_ENDIAN
```

8. Open

```
$OPENDDS_ROOT/build/target/ACE_wrappers/ace/os_include/
    os_ucontext.h
```

comment out "`typedef int ucontext_t;`" in line 40.

9. Back to `$OPENDDS_ROOT` and run:

```
make -j4 -s.
```

After which the OpenDDS libraries will be built.

A.2 Using OpenDDS Libraries in Android Development

After building OpenDDS libraries under Android configurations, all the generated libraries are static libraries (`.a` files). However, only the dynamic libraries (`.so` files) can be packaged in `.apk` file for installing the application.

The OpenDDS is compiled with Android NDK standalone toolchain since it has complicated `GNUMakefile`. Different from building OpenDDS libraries, we need use `ndk-build` to build dynamic libraries for Android applications. An `Android.mk` along with `Application.mk` is needed in the building. To create such `.mk` files and use `ndk-build`, following steps need to be taken.

1. Identify which static libraries are used to build the OpenDDS. These libraries can be found in `GNUMakefile.*` in the application folder. Search for `LDLIBS` in the `GNUMakefiles`. List them in the `Android.mk`.
2. Add necessary `CPPFLAGS` and `LDFLAGS`. Just follow it in the sample.
3. Add source files. Notice that files generated by OpenDDS IDL compiler are also necessary.
4. Add necessary platform information to `Application.mk`.

It is crucial to add all the necessary static libraries generated from standalone toolchain without omitting any one. To use the dynamic library in the application development, follow the steps listed below.

1. Use SWIG [1] to generate JNI interface automatically. Do not generate JNI interface yourself, it is error-prone.
2. Add generated `.cpp` file into `Android.mk`.
3. Add `Android.mk` and `Application.mk` to the Android Studio project. Run `Build` in Android Studio. The `ndk-build` will run, and the dynamic library file will be automatically generated under:

```
$YOUR_ANDROID_PROJECT/app/libs/$YOUR_ABI_VERSION
```

4. Utilise `.java` files which generated by SWIG in Android Studio.
5. The `.so` file will automatically be packaged in the `.apk` file.

Notice internet permission needs to be added in the Android application's `AndroidManifest.xml` file, otherwise, the application will collapse.

[1] http://www.swig.org/

Bibliography

ArduPilot Dev Team (2016a), ArduPilot Autopilot Suite.
http://ardupilot.org/ardupilot/index.html

ArduPilot Dev Team (2016b), Transmitter configuration of Copter.
http://ardupilot.org/copter/docs/common-radio-control-calibration.html

Blais, C. (2016), Unmanned systems interoperability standards, Technical report.
https://calhoun.nps.edu/handle/10945/50386

Eckerson, W. W. (1995), Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications., **vol. 10**, no.1.

Google Inc. (2017a), The Android Source Code.
https://source.android.com/source/

Google Inc. (2017b), Protocol Buffers Developer Guide.
https://developers.google.com/protocol-buffers/docs/overview

Google Inc. (2017c), Services.
https://developer.android.com/guide/components/services.html

Google Inc. (2017d), Standalone Toolchains.
https://developer.android.com/ndk/guides/standalone_toolchain.html

GPS World (2016), Commercial drone market to grow at CAGR 27 percent to 2021.
http://gpsworld.com/commercial-drone-market-to-grow-at-cagr-27-to-2021/

Meier, L., D. Honegger and M. Pollefeys (2015), PX4: A Node-Based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms, in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*.
http://ieeexplore.ieee.org/document/7140074/

Meier, L., D. Honegger and M. Pollefeys (2017), Radio Control of PX4.
https://pixhawk.org/peripherals/radio-control/start

Mohamed, N., J. Al-Jaroodi and I. Jawhar (2008), Middleware for Robotics: A Survey, in *2008 IEEE Conference on Robotics, Automation and Mechatronics*, pp. 736–742, ISSN 2158-2181, doi:10.1109/RAMECH.2008.4681485.
http://ieeexplore.ieee.org/document/4681485

Murphy, R. R. (2014), *Disaster Robotics*, MIT Press.
https://books.google.nl/books?id=9s_MAgAAQBAJ

Object Computing, Inc. (2017a), OpenDDS Building.
http://opendds.org/documents/building.html

Object Computing, Inc. (2017b), OpenDDS Developer's Guide.
http://opendds.org/documents/

Object Computing, Inc. (2017c), OpenDDS website.
http://opendds.org

Object Management Group, Inc. (2017), Data Distribution Service (DDS) Specifications.
http://www.omg.org/spec/DDS/

Oracle (2017), Java Native Interface Specification.
http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html

Pardo-Castellote, G. (2010), UAV Data Link Design for Dependable Real-Time Communications.
https://www.slideshare.net/GerardoPardo/uav-data-link-design-for-dependable-realtime-communications

PrismTech (2017), Quality of Service.
http://download.prismtech.com/docs/Vortex/html/ospl/DDSTutorial/qos.html

Shao, Y., J. Ott, Y. J. Jia, Z. Qian and Z. M. Mao (2016), The Misuse of Android Unix Domain Sockets and Security Implications, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, New York, NY, USA, CCS '16, pp. 80–91, ISBN 978-1-4503-4139-4, doi:10.1145/2976749.2978297.
http://doi.acm.org/10.1145/2976749.2978297